UNIVERSITÉ DE MONTRÉAL

ANALYSE, IMPLANTATION ET INTÉGRATION D'UNE BIBLIOTHÈQUE POUR LA SPÉCIFICATION DES SYSTÈMES EMBARQUÉS DANS UNE MÉTHODOLOGIE DE CODESIGN

LUC FILION
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME
DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)

DÉCEMBRE 2002

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

«ANALYSE, IMPLANTATION ET INTÉGRATION D'UNE BIBLIOTHÈQUE POUR LA SPÉCIFICATION DES SYSTÈMES EMBARQUÉS DANS UNE MÉTHODOLOGIE DE CODESIGN»

présenté par : FILION Luc

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. SAVARIA Yvon, Ph.D., président
M. BOIS Guy, Ph.D., membre et directeur de recherche
M. ABOULHAMID El Mostapha, Ph.D., membre et codirecteur de recherche
M. BOYER François-Raymond, Ph.D., membre

Remerciements

Je tiens d'abord à remercier mon directeur de recherche Guy Bois pour l'enthousiasme qu'il a apporté à ce projet. M. Bois aura mis beaucoup de temps, de mots, d'insistance, de confiance et d'encouragement pour le développement de la bibliothèque. Sans son soutien personnel, ce projet n'existerait pas. J'aimerais également remercier mon codirecteur Mostapha Aboulhamid, qui répondit avec intérêt à mes nombreuses questions. Je remercie inéluctablement les gens qui m'ont subventionné, d'abord mon directeur de recherche Guy Bois, puis le ReSMiQ pour la bourse qu'il m'a octroyée. Je me vois également fort reconnaissant envers mes collègues qui ont directement aidé à la réalisation de ce projet, à la rédaction d'articles ou qui m'ont personnellement supporté : Marc Bertola, Jérôme Chevalier, Geneviève Cyr, mais aussi les autres étudiants du CIRCUS et membres du personnel qui m'ont apporté soutien, conseils et sourires. Certaines gens ont contribué d'une façon intéressante à ce mémoire en travaillant sur des projets de fins d'études au baccalauréat et je les remercie de bonne foi : Jean-Philippe Richer et Hugo Lefrançois.

Et dans ces moments les plus décourageants, que reste-t-il donc? Ma chère copine Laurence qui ne cessa de m'encourager. Mais encore, mes fabuleux brownies devant lesquels je m'incline et qui m'ont manifestement revigoré à plusieurs reprises! C'est avec joie que je vous les recommande !

Résumé

Les méthodes utilisées aujourd'hui pour la conception de systèmes embarqués deviennent de moins en moins efficaces à cause de la complexité grandissante des circuits. Pour éviter une diminution de la productivité lors de la spécification des systèmes, un mouvement vers les langages orientés objets est essentiel. C'est avec des concepts tels la réutilisation et le raffinement progressif des spécifications que cette diminution sera évitée. Quelques groupes de recherche et compagnies ont opté pour cette voie, mais les bibliothèques disponibles sur le marché ne répondent pas à tous les besoins. Il est important que les concepteurs de systèmes comprennent bien ces nouvelles idées parce qu'ils feront face à un changement drastique dans leurs méthodes de design.

Nous avons d'abord revu l'entrée des spécifications dans les méthodologies existantes et avons examiné les langages et bibliothèques disponibles afin d'établir une liste des lacunes qu'elles possèdent. De là, nous avons proposé une méthodologie de conception des systèmes embarqués (ou méthodologie de codesign) qui utilise pour spécifications une bibliothèque système appelée *Syslib*. Cette bibliothèque, couvrant un niveau d'abstraction sans notion de temps (*untimed functional*), est programmée en C++ orienté objet pour faciliter le raffinement progressif vers des solutions finales d'implantation (matériel ou logiciel). Pour démontrer les possibilités de Syslib, un exemple de routeur de paquets producteur/consommateur est présenté. Nous discutons ensuite des techniques de raffinement. La spécification vers le matériel se fait à l'aide de la bibliothèque de modélisation Cynlib. Un exemple illustrant les difficultés de cette technique est présenté. Pour le raffinement vers le logiciel, on réutilise au maximum les spécifications Syslib.

Une fois la bibliothèque Syslib de niveau fonctionnel terminée, nous avons procédé à l'implantation de différents exemples de design pour démontrer les possibilités de la bibliothèque, question de bien la situer parmi l'ensemble des bibliothèques existantes. Les exemples présentés comprennent un décodeur JPEG, un contrôleur mémoire, un

BlockMatcher, un additionneur simple, de même que le routeur de paquets. Tous ces exemples ont été programmés à un niveau d'abstraction fonctionnel sans notion de temps en utilisant Syslib et SystemC, la bibliothèque « standard » de l'industrie. Les exemples sont comparés de façon quantitative mais aussi qualitative. Une analyse des résultats montre que la bibliothèque Syslib est plus performante pour des échanges de données intensifs. Aussi, les fichiers produits sont beaucoup plus petits et mieux adaptés pour l'embarquement des spécifications sur une plate-forme cible. De plus, nous avons démontré que Syslib est tout aussi simple à utiliser que SystemC (ou toute autre bibliothèque).

Les résultats de cette recherche démontrent que si l'on se dirige vers le développement d'un système en utilisant une méthodologie à raffinement progressif, Syslib apparaît comme un choix intéressant qui diminue le temps de conception d'un système.

Abstract

The embedded systems design methods in use today are becoming less and less efficient because of the growing complexity of integrated circuits. A movement towards high level and object-oriented languages (C++ for instance) for system design is necessary to avoid productivity loss. It is by supporting concepts such as component re-using and the progressive refinement of the specifications that this productivity loss will be prevented. Several research groups and companies chose this way, but the libraries available on the market do not fulfil all the needs. Moreover, system designers will have to better understand these new ideas to cope with drastic changes in design methodologies and languages.

First of all, we review the specification capture phase of many existing methodologies and looked at the available languages and programming libraries to list their features and gaps. From this point, we propose a methodology for embedded systems design (or co-design methodology), which integrates a system-level library we called *Syslib*. The Syslib library is programmed in object-oriented C++ to facilitate progressive refinement towards a final implementation solution (hardware or software). Syslib covers the untimed functional level of abstraction. To demonstrate Syslib's possibilities, a detailed example of a producer-consumer system called *PacketRouter* is explained. Then, we discuss about progressive refinement techniques. Hardware refinement is accomplished with the use of the modelling library Cynlib. For software aspects, we re-use the maximum of the Syslib specifications.

Once the Syslib library implemented, we developed various examples of design to situate the library amongst those existing. The examples presented include a JPEG decoder, a memory controller, a block matcher, a simple adder and finally the packet router presented above. To obtain comparative results (quantitative and qualitative), all these examples have been programmed at an untimed functional level of abstraction using

Syslib and SystemC (*de facto* industry standard). The results show that the Syslib library is more powerful with data-oriented applications and that the generated files are much smaller and more adapted for embedding at the architectural level. Moreover, we noticed that Syslib is as simple to use as SystemC (or others).

The results of this research demonstrate that the move towards system development using a progressive refinement methodology and a library like Syslib appears to be an interesting choice, which decrease the time of design.

Table des matières

Liste des tableaux

Liste des figures

Liste des acronymes

AC	*Alterning Current* ou courant alternatif
AHB	*Advanced High-performance Bus*
AMBA	*Advanced Microcontroller Bus Architecture*
APB	*Advanced Peripheral Bus*
API	*Application Programming Interface*
ASB	*Advanced System Bus*
ASIC	*Application Specific Integrated Circuit* ou circuit intégré dédié
BCA	*Bus Cycle Accurate*
BL	*Behavioural Level* ou niveau comportemental
CBC	*Constraint-Based Codesign*
CDFG	*Control Data Flow Graph*
CFSM	*Codesign FSM*
DC	*Direct Current* ou courant continu
DCT	*Discrete Cosine Transform*
DE	*Discrete Event*
DPRAM	*Dual Port RAM*
DSP	*Digital Signal Processor*
ESC	*Extended SystemC*
FIFO	*First In First Out*
FL	*Functional Level* ou niveau fonctionnel
FLI	*Foreign Language Interface*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
HCFSM	*Hierarchical Concurrent FSM*
HDL	*Hardware Description Language*
IP	*Intellectual Property*
ISS	*Instruction Set Simulator*

JPEG	*Joint Photographic Experts Group*
KPN	*Kahn Process Network*
L–L	Logiciel-Logiciel
L–M	Logiciel-Matériel
MCU	*Minimal Component Unit*
MH	Module hiérarchique
M–L	Matériel-Logiciel
M–M	Matériel-Matériel
MOC	*Model of Computation*
MPEG	*Moving Picture Expert Group*
MPN	Module de premier niveau
OCL	*Object Constraint Language*
OO	*Object Oriented* ou orienté objet
OSCI	*Open SystemC Initiative*
PCA	*Pin Cycle Accurate*
PLI	*Programming Language Interface*
PSP	*Processor Support Package* ou paquetage processeur
RAM	*Random Access Memory*
RGB	*Red Green Blue*
ROM	*Read-Only Memory*
RPC	*Remote Procedure Call*
RTL	*Register Transfer Level*
RTOS	*Real-Time Operating System*
SDF	*Static DataFlow*
SLDL	*System-Level Design Language*
SoC	*System-on-Chip*
SR	*Synchronous Reactive*
SRAM	*Static RAM*
STL	*Standard Template Library*

Syslib^{BL}	*Syslib **B**ehavioural **L**evel* ou Syslib niveau comportemental
Syslib^{FL}	*Syslib **F**unctional **L**evel* ou Syslib niveau fonctionnel
Syslib^{LL}	*Syslib **L**ower **L**evel* ou Syslib de niveau architectural
TarP	***Tar**get **P**latform*
TF	***T**imed **F**unctional*
UML	***U**nified **M**odeling **L**anguage*
USB	***U**niversal **S**erial **B**us*
UT	Unités de temps
UTF	***U**ntimed **F**unctional*
VC	***V**irtual **C**omponent*
VCI	***V**irtual **C**omponent **I**nterface*
VHDL	***V**HSIC **H**ardware **D**escription **L**anguage*
VHSIC	***V**ery **H**igh **S**cale **I**ntegrated **C**ircuit*
VSIA	***V**irtual **S**ocket **I**nterface **A**lliance*
YUV	**Y** = luminance, **U** = bleu-Y, **V** = rouge-Y (YUV ≅ 0,3·Rouge + 0,6·Vert + 0,1·Bleu)

Lexique

Bibliothèque	Assortiment logiciel regroupant des services pour fins de programmation. En français, on dira souvent, à tort par anglicisme, librairie. **Anglais :** library
Binding	Opération d'association des ports aux canaux ou à des ports hiérarchiques. **Français :** association.
Burst Mode	Mode de transmission de données sur un bus dont le protocole est d'envoyer une seule adresse pour une série de données. **Français :** mode rafale ou transaction éclatée.
Byte enable	Sur une mémoire, signal servant à la sélection du bon octet.
Chip select	Sur une mémoire, signal servant à l'activation de celle-ci.
Chrominance	En télévision, signaux portant la composition de couleur d'une image.
Codesign	Art de concevoir des systèmes logiciel-matériel fortement liés, des systèmes embarqués ou des systèmes sur puce. **Français :** co-spécification est une traduction correcte, mais comme « design » est reconnu en français, on emploiera « codesign ».
Datapath	**Français :** chemin de données.
Espace colorimétrique	**Anglais :** color space.
Event-driven	Contrôlé par un événement, par un signal.
Flag	Variable indiquant un état particulier. **Français :** drapeau.
Handler	Sous-programme qui permet la gestion de certaines opérations à l'intérieur du système. **Français :** gestionnaire.

Handshaking	Protocole de communication nécessitant au cours duquel deux entités doivent échanger des signaux, jusqu'à ce qu'ils soient parfaitement synchronisés et prêts à communiquer entre eux. **Français :** protocole de poignée de mains.
Lookup Table	Tableau de références contenant des valeurs pré-calculées permettant d'établir une relation entre deux séries de valeurs. **Français :** table de conversion.
Luminance	En télévision, signaux décrivant la distribution de luminosité d'une image.
Macro	Séquence d'instructions enregistrée sous un nom, qu'on peut rappeler et exécuter par le nom qui lui a été attribué.
Map	En STL, structure de données de type conteneur associatif qui permet de lier à tout objet qui y est inséré une clef unique. **Français :** ensemble.
Micronoyau	Système d'exploitation simplifié n'offrant que les concepts de bases. **Anglais :** Microkernel.
multi-thread	À plusieurs processus élémentaires.
Mutex	Sémaphore binaire.
Netlist	Énumération des connexions entre les différents composants ou blocs fonctionnels d'un circuit. **Français :** liste d'interconnexions.
Output enable	Sur une mémoire, signal servant à valider les sorties de lecture.
Parser	Outil qui analyse des textes à partir de la reconnaissance d'unités syntaxiques à partir d'une grammaire hors contexte. **Français :** analyseur syntaxique.
Pipeline	Architecture structurelle qui permet de séparer un traitement en plusieurs étages d'exécution et de recouvrir simultanément l'exécution de chaque étage.
Pixel	Élément graphique élémentaire caractérisé par une couleur et une intensité.

Polling	Mode de traitement dans lequel l'entité chargée du contrôle des échanges sur un réseau interroge régulièrement chaque élément branché, afin de savoir si elle a des données à émettre ou si elle est prête à en recevoir. **Français :** interrogation, scrutation.
Port-Socket	Voir *socket*. **Français :** capsule
Préemption	Mode d'opération d'ordonnancement d'un système d'exploitation qui permet de retirer une tâche en exécution pour la remplacer par une autre plus prioritaire.
Reset	Remise d'un processus ou d'un système dans son état standard de démarrage. **Français :** réinitialisation.
Socket	Interface de connexion ou connecteur logiciel permettant de communiquer d'une entité à une autre.
Template	En C++, superclasse définissant un composant logiciel fortement paramétré. **Français :** Modèle ou classe générique.
Testbench	Ensemble des tests utilisés pour valider une spécification. **Français :** banc d'essai.
Thread	Processus élémentaire.
Timer	Dispositif qui permet aux entités d'établir les intervalles de temps désirés pour divers tâches. **Français :** temporisateur.
Timing	Correspondance exacte de temps entre des impulsions servant à la synchronisation. **Français :** minutage.
Untimed functional	Niveau fonctionnel sans notion de temps ou sans minutage.
Watchdog	Dispositif qui surveille un signal afin de déceler l'absence d'une action à la fin d'une période déterminée pour effectuer un traitement compensatoire. **Français :** chien de garde.

Watching (local, global)	Surveillance des signaux servant à l'établissement d'un système de traitement d'exceptions. **Français :** surveillance, horloge de surveillance.
Wrapper	Entité permettant d'en adapter une autre pour assurer une compatibilité au niveau des échanges de données. **Français :** enveloppe, adaptateur.
YUV	Espace colorimétrique vidéo utilisé dans la télévision.

Liste des annexes

Introduction

Un système est qualifié d'embarqué lorsqu'il est dédié à une tâche informatique spécifique dans un environnement donné. Souvent, les systèmes embarqués seront implantés sur une puce unique que l'on nomme *system-on-chip* ou *SoC*. Ces systèmes sont séparés en deux catégories distinctes: des systèmes orientés pour le contrôle et des systèmes orientés pour le traitement des données. Des systèmes peuvent très bien être un mélange des deux. Des exemples de chacune de ces catégories sont donnés au tableau 1.

Tableau 1 : Exemples de systèmes embarqués

Systèmes orientés contrôle	Systèmes orientés traitement de données
• Contrôleur de freins ABS	• Décodeur vidéo MPEG4
• Four à micro-ondes	• Téléphonie cellulaire
• Contrôleur de lance-missiles sur un chasseur	• Commutateur dans les grands réseaux Internet

Que les systèmes soient contraints à opérer dans des environnements particuliers apporte son lot de difficultés. Par exemple, la consommation de puissance de certains de ces systèmes doit être réduite. Le cas du téléphone cellulaire en est un exemple évident. Par ailleurs, le système doit également répondre en temps réel aux stimuli externes auxquels il est soumis. Le temps réel indique qu'un système doit fournir une réponse aux stimuli dans un temps suffisamment petit et prévisible pour répondre au monde physique qui l'entoure. De plus amples informations et exemples sont présentés dans [BuWe97].

Techniquement, un système embarqué comprend une partie « matérielle », composée de circuits logiques dédiés ou ASIC, de DSP et FPGA, de mémoires. Également, ces systèmes sont munis d'une partie « logicielle » composée de programmes stockés en ROM s'exécutant sur des microprocesseurs dédiés. Des bus et interfaces de communication relient les deux mondes. Un exemple complexe de système embarqué est donné à la figure 1 où l'on voit la structure d'une caméra numérique.

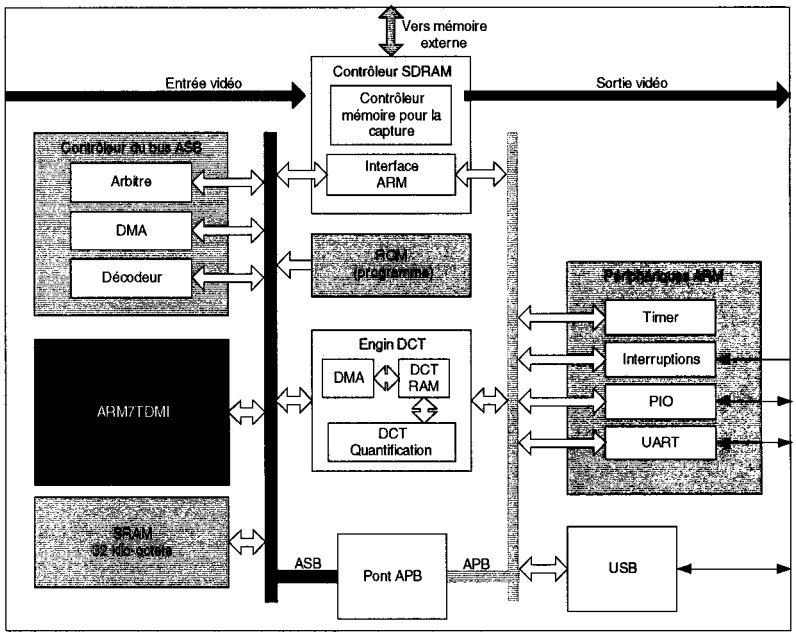


Figure 1 : Exemple classique d'un système embarqué

Dans cet exemple, nous voyons à gauche, le microprocesseur de type ARM7TDMI branché à un bus normalisé AMBA (ASB). À ce même bus sont reliés des mémoires SRAM (qui contiennent entre autre le programme à exécuter), des contrôleurs ainsi que des circuits logiques dédiés, comme l'engin DCT. Un bus de périphériques à droite (APB) permet de brancher le système au monde extérieur (par exemple un magnétoscope).

Le codesign se définit comme l'art de concevoir des systèmes sur lesquels le logiciel et le matériel sont intimement liés. On définira une méthodologie de codesign comme les multiples étapes nécessaires à la conception d'un SoC ou un PCB. En voici les principales étapes (tirées de [BCGH97]) représentées à la figure 2. On débute généralement avec la capture de la spécification grâce à des langages de haut niveau. On valide la spécification à l'aide de plusieurs simulations. Une fois la spécification validée, le concepteur passe à l'étape du partitionnement logiciel/matériel : il identifie les parties d'un système qui seront implantées en logiciel et quelles autres le seront en matériel. Une

façon formelle de parvenir à cette fin consiste en l'utilisation d'estimateurs, qui calculeront les coûts du logiciel (taille du code, temps d'exécution, surcoût rattaché au système d'exploitation, etc.) et les coûts du matériel (nombre de portes logiques, dissipation de puissance, nombre d'entrées-sorties, etc.) Une fois le partitionnement complété, on procède à un raffinement du matériel et du logiciel. Une fois les modules raffinés, on passe à la simulation conjointe du logiciel et du matériel ou co-simulation. Finalement, le système passe par une étape de synthèse pour transformer le comportement du matériel en circuit synthétisable (*netlist*) et pour compiler le logiciel (code machine) qui sera placé en ROM sur l'architecture. Il s'agit ensuite de valider les prototypes générés et de fabriquer la version finale du système sur une puce physique. Il va de soi que, si l'une des étapes révèle une insatisfaction en terme de performance ou de comportement, les concepteurs peuvent revenir en arrière.

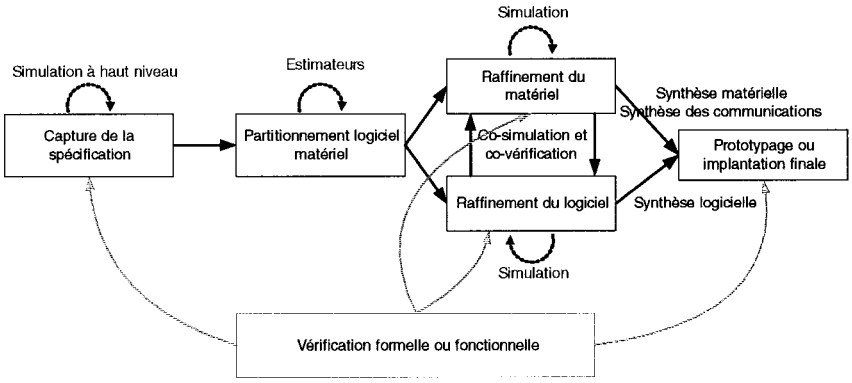


Figure 2 : Méthodologie générale de codesign

La tendance veut qu'à plusieurs (voire toutes) étapes de la méthodologie, on procède à une vérification fonctionnelle du système spécifié. La vérification fonctionnelle tente de prouver que le système programmé est bel et bien celui qui a été spécifié. Elle se sépare en deux catégories complémentaires. Il y a la vérification formelle, qui utilise des modèles théoriques et mathématiques représentant la spécification pour prouver les aspects fonctionnels d'un système. La vérification fonctionnelle par simulation, quant à elle, cherche entre autre à balayer le plus de chemins d'exécution possibles (lors de

simulations) du système en design pour en vérifier la fonctionnalité. La vérification conjointe du matériel et du logiciel se nomme co-vérification. Le présent mémoire ne couvre pas les aspects de vérification, mais pour plus d'information, le lecteur peut se référer à [VSIA01], [RPS01] et [Berg00].

Problématique

Depuis quelques années, l'étape de la capture des spécifications est de plus en plus réalisée à l'aide de langages de spécification de niveau d'abstraction élevé : C, C++, Java, etc. dont certains offrent le concept de l'orienté objet. Ces langages requièrent malheureusement une transformation syntaxique et comportementale complète pour passer aux étapes de codesign subséquentes. Certains groupes étendent ces langages en des bibliothèques permettant la conception orientée système, afin d'abstraire les éléments d'un système embarqué pour effectuer des simulations à un niveau le plus élevé possible. Dans certains cas, il est possible d'effectuer des simulations de matériel grâce à ces bibliothèques, ce qui diminue le temps de raffinement des spécifications et augmente la performance des simulations. Par exemple, des bibliothèques C++ reconnues telles SystemC [Swan01], SpecC [GZDG00] et Cynlib [FDS00a] se révèlent comme des propositions intéressantes. Ces bibliothèques, liées à une méthodologie solide, deviennent des outils puissants de conception.

Toutefois, ces bibliothèques existantes ne répondent pas tout à fait aux besoins de conception des SoC sous plusieurs aspects. D'abord, très peu d'entre elles se lient à une méthodologie complète, ce qui ampute leur efficacité; tel est le cas de SystemC présentée à la figure 3 qui ne schématise que grossièrement les directions à prendre pour concevoir un système et qui ne propose aucune solution concrète pour le partitionnement, la description du logiciel, les liens avec un système d'exploitation, ainsi que les communications entre les modules matériel et logiciel. Des versions futures sont cependant attendues.

SpecC propose quant à elle une méthodologie complète pour les SoC. Cependant, un problème majeur surgit: l'utilisation d'un langage non standardisé limite son utilisation aux seuls outils fournis par SpecC, à moins qu'on développe ses propres outils, ce qui est un travail long et fastidieux. Finalement, Cynlib propose un support flexible et simple pour le matériel. Toutefois, aucun support pour le logiciel ou aucune vision système n'est supportée, ce qui restreint ses possibilités.

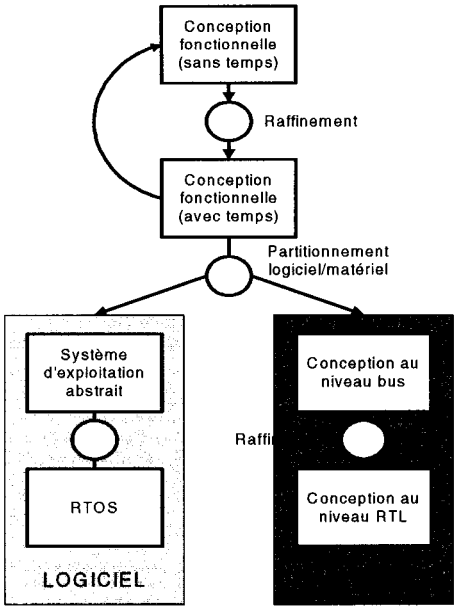


Figure 3 : Méthodologie de conception de SystemC

Objectifs

L'objectif principal de ce travail est d'intégrer une bibliothèque orientée objet de conception de systèmes embarqués dans une méthodologie complète de codesign. Définir une bibliothèque à tous les niveaux est un travail ardu et c'est pourquoi il importe de s'inspirer des travaux existants. Comme Cynlib n'est utile que pour la modélisation de matériel, on y superposera Syslib, notre bibliothèque de niveau système, en C++ orienté objet, qui complémentera les besoins système de Cynlib. Il importe également de prouver qu'on peut utiliser sans tracas la bibliothèque Syslib pour programmer des exemples de design. Enfin, on voudra s'assurer que la bibliothèque programmée est performante et efficace en la comparant à une bibliothèque existante, en l'occurrence SystemC.

Méthodologie

Il faut établir les besoins de la bibliothèque Syslib pour qu'elle puisse bien s'intégrer à une méthodologie de codesign. Une méthodologie complète est effectivement complexe et il apparaît évident que seulement quelques étapes seront couvertes dans ce mémoire. La bibliothèque sera implantée à un niveau d'abstraction élevé sans notion de temps (voir figure 3). Les autres étapes feront parti de travaux futurs au sein du CIRCUS[1]. Pour vérifier le bon fonctionnement de la bibliothèque, nous avons développé des exemples de design pouvant être capturées à haut niveau selon différents types de traitements (orienté contrôle, flôt de données, etc.). Il est par le fait même important de démontrer que le raffinement vers Cynlib se fait sans encombre et nous en donnons un exemple simple. Finalement, mentionnons qu'au cours de la réalisation de ce projet, SystemC sortit une version 2 de sa bibliothèque couvrant les aspects système en plus des aspects de matériel. Une comparaison entre les possibilités et performances de Syslib et SystemC sera donc présentée.

Originalité et contribution

L'originalité de ce travail est démontrée par l'avantage d'avoir Syslib à sa portée. Cette bibliothèque fournit tous les outils nécessaires à la modélisation rapide d'un système embarqué. Cynlib ne possède aucun niveau système et puisque aucun n'est prévu par ses concepteurs, la combinaison Syslib/Cynlib est une solution intéressante pour la conception de systèmes. Cette liaison est supportée par la méthodologie de codesign proposée, ce que très peu de bibliothèques proposent. Syslib utilise également une approche de conception différente en proposant des concepts intéressants qu'on ne retrouve pas à ce jour dans les autres bibliothèques.

---

[1] Le CIRCUS est un sous-groupe du GRM (Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal) qui s'intéresse à la recherche sur le codesign.

De plus, ce travail est accompagné d'un outil graphique se nommant Picasso, permettant de faciliter l'entrée des spécifications, entre autre en générant automatiquement les connexions d'un système complexe. L'utilisation de moyens graphiques facilite le travail du concepteur en réduisant le temps nécessaire à l'entrée des spécifications [HePa96]. Il n'est cependant pas obligatoire d'utiliser Picasso pour l'entrée des spécifications. Syslib est indépendant de Picasso.

Également, ce projet a mené à la publication de plusieurs articles. D'abord [FiBA02a] a présenté notre méthodologie de codesign et l'outil Picasso au 70^{e} congrès de l'ACFAS. Ensuite, [FiBA02b] a présenté la bibliothèque Syslib ainsi qu'un exemple de design à la 11^{e} conférence HDLCon. Enfin, [FCBA02] a expliqué la méthodologie utilisée pour le design de la bibliothèque et l'engin de simulation au 2^{e} atelier IWSOC. Toutes ces publications sont disponibles sur le site Internet du laboratoire de recherche sur le codesign de l'École Polytechnique, le CIRCUS [CIRC02].

Distribution des chapitres

Ce mémoire est distribué en quatre chapitres. Le chapitre 1 survole les méthodologies de conception des systèmes embarqués, les langages ou bibliothèques de haut niveau utilisés pour développer des systèmes et l'apport de l'orienté objet dans la conception de systèmes. Le chapitre 2 présente la bibliothèque Syslib et sa méthodologie de codesign, son implantation et un exemple de système. Les problèmes rencontrés et les solutions choisies y sont aussi présentés. Le chapitre 3 traite de raffinement, une étape essentielle lors de tout design de système. Nous verrons les endroits où le raffinement s'impose et les principales étapes à suivre. Enfin, le chapitre 4 présente les résultats obtenus de quelques exemples de design de systèmes ainsi qu'une comparaison de ces mêmes exemples programmés avec SystemC.

CHAPITRE 1
Revue de la conception des SoC

Les systèmes embarqués s'intègrent dans de plus en plus d'applications de la vie quotidienne. Ce nouveau marché en croissance mènera l'industrie à redéfinir ses méthodes et techniques de conception pour répondre à l'extraordinaire demande qui surviendra aux cours des prochaines années. Pour mieux comprendre vers où les méthodes de conception se dirigent, on se doit également de comprendre d'où elles proviennent. Ce chapitre survole plusieurs aspects des outils, approches et méthodologies de conception existants des systèmes embarqués. Depuis des années, ces techniques et technologies ont évolué et ont progressé vers l'abstraction des descriptions, rendant possible la manipulation des systèmes complexes d'aujourd'hui. Les méthodologies les plus importantes et les plus pertinentes seront présentées. Nous verrons donc une partie des plus récents travaux en matière de spécification des systèmes embarqués. Nous nous attarderons particulièrement à la toute nouvelle vague de l'utilisation de l'orienté objet pour définir des systèmes en étudiant les principaux langages et bibliothèques de programmation qui attirent les industries et les groupes de recherche.

1.1. Approches de conception

Les ingénieurs de l'industrie utilisent plusieurs approches ou empruntent plusieurs voies pour concevoir un système embarqué. Une première voie consiste à intégrer des spécifications de matériel et de logiciel validées sur une plate-forme d'exécution. La plate-forme existe, pour fins de simulation, en langage de modélisation de matériel, tels VHDL ou Verilog. Le logiciel, sous forme de bibliothèques compilées, s'exécute sur un processeur embarqué. Il s'agit de l'approche plate-forme très prisée et diversifiée pour servir des marchés distincts. Une seconde approche consiste à modéliser un système complet au moyen d'un langage de haut niveau incluant les spécifications de l'application et la plate-forme d'exécution. Une simulation de haut niveau est effectuée pour valider le comportement du système, puis on raffine ensuite la spécification vers des niveaux

d'implantation, mais en conservant le même langage : c'est l'approche par raffinements progressifs. Dans les deux cas, une fois le système complètement validé, on passera à la synthèse du matériel, du logiciel et des communications. Il est évident qu'un mélange de ces deux approches est légitime.

1.1.1. Approche par raffinement progressif

Cette approche est amenée par un langage spécialisé ou encore une bibliothèque de programmation qu'un programmeur inclut dans son code. Il est souvent question de langage orienté objet, par exemple C++, qui offre beaucoup de flexibilité pour ce genre de pratique.

Parmi les approches à raffinements progressifs, une d'entre-elles consiste à transformer des spécifications fonctionnelles vers des modèles architecturaux prédéfinis, principalement pour les communications et les ressources disponibles. Ces raffinements se tiennent dans les étapes de partitionnement et d'allocation des ressources [GoGB97]. Cette approche offre peu de flexibilité, par contre, la simplicité et l'efficacité du design obtenu est un avantage assuré.

Une autre solution consiste à utiliser des bibliothèques de conception orientées objet qui permettent de décrire un système complet à plusieurs niveaux d'abstraction et de raffiner les spécifications jusqu'au niveau final d'implantation [Swan01]. Les spécifications débutent à un niveau fonctionnel et sont raffinées vers un niveau matériel tout en conservant la même bibliothèque. Les concepteurs ont le choix ou non d'inclure les détails de l'architecture dans les spécifications pour obtenir des résultats plus variés. Les possibilités sont infinies, mais le tout doit se faire manuellement.

1.1.2. Approche plate-forme

On utilise souvent le terme architecture d'un système ou seulement architecture pour décrire l'organisation interne du système, i.e. la structure qui le compose. Le terme ne doit pas être confondu avec le concept d'architecture d'un ordinateur (ou d'un

microprocesseur) qui correspond non seulement à sa structure, mais également à son jeu d'instructions. Dans [HePa96], on différencie longuement ces concepts. Le terme plate-forme ne représente donc pas l'architecture du système en soi, mais plutôt une structure de base utilisée par les ingénieurs pour construire une nouvelle application de système embarqué. La grande majorité des compagnies qui conçoivent des SoC réutilisent en tout ou en partie la plate-forme d'un produit précédent, ce qui diminue le temps de conception du nouveau produit par la réutilisation de plusieurs composants ou modules déjà validés.

On parlera souvent de conception orientée plate-forme *(Platform-Based Design)* dans la littérature [CCHM99]. Cette solution permet de modéliser un SoC à partir d'une plate-forme d'exécution sur laquelle on insérera des blocs prédéfinis en fonction de l'application à concevoir. Une multitude de systèmes sont possibles, par contre, une normalisation des interfaces de communication est requise, de même que la disponibilité des blocs à insérer respectant cette norme.

Des compagnies proposent d'intégrer leurs microprocesseurs sur des plates-formes qu'ils ont eux-mêmes conçues. Par exemple, la compagnie ARM [ARM00a], fabricant d'une famille de microprocesseurs dédiés aux systèmes embarqués, propose une plate-forme basée sur une norme de bus nommée AMBA [ARM00b] sur laquelle plusieurs bus et protocoles de communication s'adaptent à tout design. Également, IBM propose sa plate-forme CoreConnect adaptée à son processeur PowerPC [BeLe00]. Plusieurs outils sont fournis avec la plate-forme ce qui en fait un produit intéressant pour les ingénieurs. D'autres produits commerciaux basés sur l'approche plate-forme existent, notamment Platform Express de Mentor Graphics [Ment02a]. Le but de cet outil est de créer rapidement une plate-forme à l'aide d'une banque de blocs prédéfinis (processeurs ARM7, ARM9, mémoire, DMA, USB, IP de tiers partis) basée sur les produits ARM et la norme de bus AMBA. Une fois la plate-forme générée, il est possible de la simuler avec les outils de Mentor Graphics.

1.2. Sémantique et normalisation

L'importance d'avoir une définition exacte pour tous les composants d'un système à tous ses niveaux est capitale. Avant de fixer des normes pour la création de système, il importe d'en établir la sémantique. La sémantique – du grec *sêmantikos* (qui signifie) – d'une chose, c'est l'étude scientifique et théorique d'un concept, qui le définit intégralement et dont toute interprétation personnelle est écartée. Puisque des milliers de compagnies et groupes de recherche développent des systèmes et composants, il s'ensuit l'établissement de normes à tous les niveaux de conception qui facilitent la compréhension et l'intégration des composants de même que leur interaction dans un système complet.

1.2.1. Accellera

Le groupe Accellera [Acce02] est composé de plusieurs industries et groupes de recherche qui ont pour but commun d'établir la sémantique dans la conception des systèmes embarqués : le tout permettra éventuellement d'établir des normes dans l'industrie. Un travail est déjà entrepris au niveau RTL [BaGa01]. Il a été appliqué à SystemVerilog [Suth02], une nouvelle extension à Verilog 2001 pour supporter la vérification et les communications de niveau architectural entre les modules. Il s'agit d'un pas vers le niveau système.

1.2.2. VSIA

Par ailleurs, une autre alliance industrielle et universitaire a déjà accompli beaucoup de travail pour normaliser la conception des systèmes. *Virtual Socket Interface Alliance* ou VSIA [VSIA02] énonce des normes pour inciter l'industrie à opter progressivement pour l'adoption d'un modèle de conception de systèmes basé sur l'exploration architecturale et l'échange de composant virtuels ou VC [VSIA97]. En d'autres termes, VSIA prône l'échange de composants protégés ou IP entre compagnies. En supposant que tous ces IP soient bâtis selon la même norme, leur intégration se fera théoriquement sans souci sérieux. VSIA propose de normaliser à trois endroits où un besoin se fait particulièrement sentir [LSJH00] :

(1) l'utilisation des principes de communication communs ;

(2) l'utilisation d'un format de design commun ;

(3) l'usage d'une approche unifiée pour analyser les performances et la qualité du système.

Une proposition pour une norme supportant l'interopérabilité est également à l'étude. L'interopérabilité permet de mélanger des VC provenant d'environnements différents [DGOS01]. Plusieurs travaux de VSIA en cours sont résumés dans ce qui suit.

System-Level Interface Behavioral (SLIF) [VSIA00a] permet de mieux gérer l'intégration des VC entre eux. Un principe de conception basé sur les interfaces s'effectue en séparant le protocole de communication des comportements internes aux VC. On peut observer ce phénomène à la figure 1.1a, alors que deux protocoles s'attachent à l'interface d'un module. On arrive ainsi à créer une orthogonalité des VC et des protocoles qu'ils utilisent. La description des VC se fait sur un ensemble de couches représentant les différents niveaux d'abstraction de ce VC. Le protocole est exprimé grâce à une série de règles de transactions basées sur le flux de contrôle et de données fonctionnant avec des tampons et des queues, bloquant ou non avec ou sans priorité.

On-Chip Bus Standard (OCB VCI) [VSIA00b] offre un ensemble de signaux logiques ainsi que des protocoles flexibles pour transférer de l'information sur un bus entre deux points. On observe à la figure 1.1b un VC branché sur un bus par l'intermédiaire d'un adaptateur (*bus wrapper*) simple pour se conformer à la norme du bus. OCB VCI offre trois niveaux de complexité selon les besoins du concepteur: un niveau de base (*Basic VCI*), un niveau avancé (*Advanced VCI*) et un niveau pour les périphériques (*Peripheral VCI*). Ces normes offrent les fonctionnalités typiques d'un bus telles les transactions séparées (*split transactions*), les rafales (*burst*), le pipelinage et l'envoi de paquets.

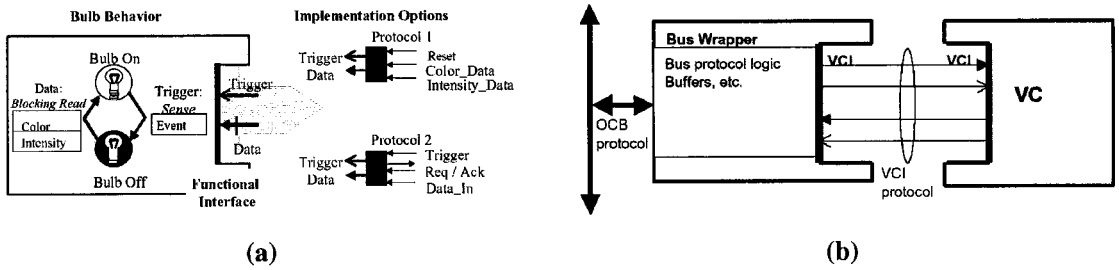


Figure 1.1 : Les concepts de normalisation proposés par VSIA

Enfin, *System-Level Data-Types Standard* [LSJH00] propose des types de données normalisés. La norme propose entre autre de ne rien supposer quant à la taille des types au niveau des compilateurs ou encore d'affecter une valeur par défaut lors de la déclaration de variables. La proposition couvre aussi bien les types de haut niveau tels les entiers signés ainsi que les types de bas niveau comme les vecteurs de bits.

1.3. Contribution du logiciel et de l'orienté objet

Les méthodologies employées pour la conception des systèmes embarqués aujourd'hui ne répondent plus et ne résisteront pas à la complexité grandissante des systèmes. La densité de transistors par puce augmente au rythme de 60% par année selon la loi de Moore [VaGi02]. Cette forte augmentation crée des systèmes fort complexes ayant des besoins accrus en terme de performance de simulation et de vérification des circuits. Les méthodes pour construire ces systèmes doivent également se renouveler, sans quoi elles deviendront des obstacles majeurs dans leur fabrication.

1.3.1. Évolution de l'utilisation du logiciel

En regardant la façon dont un système est développé, on constate trois éléments clefs où l'entrée de code est requise: à l'implantation même des algorithmes et des fonctions, à la description de l'architecture utilisée et enfin, lors de la création de plans de vérification. La figure 1.2 proposée par [DaCl02] illustre ce principe. Le triangle de gauche montre qu'au début des années 1980, trois langages (ou concepts) différents étaient utilisés pour créer un système, à l'époque, de quelques centaines de milliers de portes logiques. Avec

le temps, les ingénieurs ont découvert que l'utilisation du même langage pour la vérification et l'implantation augmentait la productivité, grâce aux mêmes outils de développement (le triangle du centre). Ainsi, on pouvait valider un système plus rapidement à un niveau plus élevé que le niveau porte logique. Puis, grâce à la synthèse logique, on pouvait transformer automatiquement la spécification Verilog vérifiée en une liste d'interconnexions (*netlist*). Comme la description de l'architecture d'un système demande un plus grand niveau d'abstraction parce que plus complexe, l'architecture à cette époque était principalement implantée en C/C++. Étant donné la complexité grandissante des systèmes (plus de 10 millions de portes logiques aujourd'hui), le C++ prendra forcément la place de Verilog/VHDL au niveau de l'implantation et de la vérification. On prédit que le langage pour décrire l'architecture de l'ère précédente prendra la place du langage d'implantation et de vérification de demain (triangle de droite). C'est par la synthèse comportementale qu'on parviendra à transformer une description de l'implantation en C++ vers les niveaux inférieurs (portes logiques). Reste à savoir quel langage ou moyen sera utilisé au cours des prochaines années pour décrire les futures architectures!

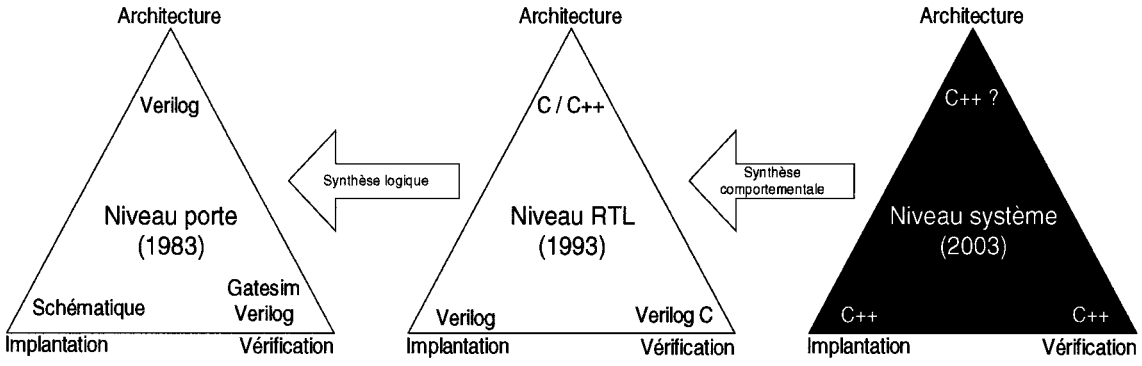


Figure 1.2 : Évolution de l'utilisation du C/C++ pour la création de SoC

Un argument supplémentaire qui supporte l'utilisation de logiciel pour décrire les systèmes est que le logiciel occupe une place de plus en plus grande dans les applications de systèmes embarqués. On retrouve sur les SoC de plus en plus de microprocesseurs

exécutant en parallèle du logiciel rapide à programmer, facile à corriger en cas de bogues tardifs et supporté par des RTOS efficaces.

1.3.2. L'orienté objet

L'orienté objet (OO), offert par des langages tels C++ ou Java répond à beaucoup d'incertitudes quant à l'avenir du design de systèmes. Le concept OO permet la **réutilisation** des modules déjà codés et vérifiés, ce qui réduit le temps de conception. La notion de **classes** est utile pour définir les éléments et modules d'un système, comme une mémoire, un processus ou encore un vecteur de bits qu'on peut manipuler. L'**héritage** sert à clarifier la définition d'une classe en y ajoutant des éléments concrets. Par exemple, on peut dériver d'une classe mémoire, des classes de mémoires en lecture seule, mémoires à deux ports d'accès, etc. Ce concept sert également au raffinement des modules où un programmeur n'a qu'à dériver un module en réécrivant les **méthodes virtuelles** qui supportent les détails correspondant à ce nouveau module. La **surcharge des opérateurs** permet de manipuler facilement des objets définis par les classes. Ainsi, on pourra additionner entre eux des vecteurs de bits sans effort. Les **modèles** (*templates*) permettent de créer des algorithmes génériques sur plusieurs types de données, par exemple, ces mêmes vecteurs de bits vs un entier pur. Toutes ces possibilités offertes par l'orienté objet sont déterminantes pour supporter un mélange de modèles de calcul et de paradigmes d'implantation (logiciel et matériel) qu'offre la conception orientée système.

Il y a plusieurs avantages à créer une version OO d'un système en conception et particulièrement d'utiliser le même langage à tous les niveaux d'abstraction précédant l'implantation finale. D'abord, la première version programmée servira de base à tout programmeur. Cela permet un raffinement simple et progressif de la spécification vers des niveaux inférieurs, ce qui réduit la propagation d'erreurs (de syntaxe ou de fonctionnalité) dans le système. La facilité de créer différents niveaux d'abstraction en C++ permet au concepteur de valider des fonctionnalités spécifiques au niveau le plus approprié. Ensuite, ces modèles préliminaires permettent de rassembler rapidement les éléments représentatifs d'un système puis d'en générer une version exécutable qu'on peut

facilement déboguer. Enfin, la simulation à très haut niveau d'abstraction (la simple exécution algorithmique d'un système) permet un gain de performance incontestable sur ce même système décrit à un niveau d'abstraction RTL. Par ailleurs, la facilité et la rapidité avec laquelle on peut décrire, en C++ orienté objet, une architecture complète soutiennent le choix de ce langage pour le développement de systèmes complexes. La contribution de l'orienté objet a lieu aussi bien au niveau spécification qu'au niveau vérification fonctionnelle. La création de bancs d'essais (manuelle ou automatisée) permet de vérifier plusieurs parties d'un système avant même de raffiner le code vers les niveaux inférieurs. L'évolution de la conception ne se base pas uniquement sur le ou les langages utilisés mais aussi sur la méthodologie dans laquelle ce langage s'intègre. Quelques méthodologies décrites dans les prochaines sections proposent des idées et façons de procéder qui épousent fort bien ces concepts orientés objets.

Malgré tout ces faits, la programmation par objet de systèmes n'obtient toujours pas l'ampleur escomptée. Cela s'explique par plusieurs facteurs. Entre autre, les programmeurs de systèmes et de matériel actuellement sur le marché ne possèdent pas l'expertise des langages de haut niveau. Il faudra par conséquent attendre que la prochaine génération de programmeurs prenne une place décisionnelle en compagnie. Également, les méthodologies de design et les outils de développement déjà en place requièrent des changements importants que les compagnies ne sont pas prêtes à assumer pour le moment étant donné le contexte économique actuel, mais aussi le statut embryonnaire de ces techniques. Bien que l'orienté objet ait déjà fait ses preuves du côté logiciel, beaucoup de recherche sur ses possibilités, ses limites et son apport au domaine de la conception de systèmes doit être accomplie avant que celui-ci ne se répande sur le marché. L'introduction d'outils commerciaux considérables, notamment VCC de Cadence (section 1.6.2) et N2C de Coware [Cowa02] n'ont eu que très peu de succès. Trop peu trop tôt dira-t-on, mais d'autres produits continuent d'envahir le marché, notamment CoCentric de Synopsys (section 1.6.4), qui sera à surveiller.

1.4. Modèles de calcul

Cette section décrit le modèle de calcul ou *model of computation* (MOC). Ces modèles représentent la philosophie à suivre derrière chaque spécification. Ils se retrouvent dans chaque langage de programmation ou à chaque niveau d'abstraction. Il importe ici de connaître leur utilité et d'identifier les plus importants.

1.4.1. Définition

Un modèle de calcul constitue une philosophie basée sur un ensemble de règles strictes qui gouvernent les interactions et le comportement d'une entité [Lee99]. On pourra également parler de sémantique d'un MOC. Spécifiquement, ces règles permettent de définir les relations entre des entités, d'établir des principes de communication, de contrôler la composition d'un design au niveau de la structure hiérarchique, de la concurrence, de l'exécution et de la représentation du temps, etc. [GLMS02].

Selon [VaGi02], si le modèle décrit le comportement du système, le langage utilisé doit savoir capturer ce modèle efficacement. Souvent, on verra un langage, une bibliothèque ou un outil orienté vers un modèle de calcul précis ou tout au plus quelques modèles. C'est notamment le cas de Ptolemy-II qui utilise Java pour spécifier de multiples systèmes basés sur des MOC différents et indépendants [DHKL00].

1.4.2. Quelques modèles connus

Il existe un bon nombre de modèles de calcul servant à de multiples fins. Le but de cette section n'est pas de les énumérer tous, mais plutôt de présenter les plus populaires dans le domaine du codesign logiciel/matériel.

En présence de systèmes plus simples et orientés pour le contrôle, l'utilisation d'une machine à états finis (FSM) est populaire et fortement recommandée. Son principe facile d'utilisation est basé sur des transitions entre des états représentant le comportement du système. Les réseaux de Pétri en sont une variation qui permet de modéliser des systèmes concurrents. Le modèle FSM est peu attrayant pour l'échange de données et c'est

pourquoi des extensions ont été proposées, dont les machines à états finis étendues (EFSM) [GVNG94], les machines à états pour le codesign (CFSM) [BCGH97] ou même une version à représentation hiérarchique de processus concurrents (HCFSM) [VaGi02].

Pour des systèmes plus complexes où s'entremêlent des échanges de données et de contrôle, un concept de graphe de flux de contrôle et de données a été élaboré (CDFG). Il constitue un modèle hétérogène qui permet de représenter les dépendances de données aussi bien que les séquences de contrôle d'une application [GVNG94].

Un autre modèle très populaire consiste à se baser sur les événements discrets (DE) se produisant dans un système. Ce modèle pour décrire des environnements matériels concurrents est intégré à plusieurs langages dont VHDL. Un autre modèle nommé synchrone/réactif (SR) est un modèle simplifié (par rapport à DE) qui possède une notion de temps global à tous les processus. Il est utilisé entre autre dans le langage Esterel [Lee99].

Enfin, pour des systèmes plus complexes, les réseaux de processus de Kahn (KPN) [Kahn74] sont excellents pour décrire des modèles algorithmiques sans notion de temps. Les échanges se font via des files d'attente infiniment longues à l'aide d'opérations bloquantes pour contrôler le flux d'exécution. Une version simplifiée des KPN, les réseaux à flux de données statiques (SDF) s'exécutent de façon suivante: lecture des entrées, exécution de calculs, propagation des sorties. Cette façon de faire permet de connaître le flot d'exécution avant l'exécution et de créer un ordonnancement statique des processus [GLMS02].

1.4.3. Modèles de performance

Il devient de plus en plus courant d'inclure des modèles de performance dans les spécifications de haut niveau pour des analyses préliminaires des performances d'un système et de son architecture. Il s'agit d'évaluer des métriques de qualité [GVNG94] reliées aux coûts du logiciel et du matériel. Il ne s'agit pas seulement de calculer le

nombre de lignes de code ou d'évaluer l'aire finale de la puce matérielle : on voudra obtenir des estimations justes au niveau logiciel telles que le nombre de tâches pour le processeur et leur ordonnancement, l'allocation des registres, les chargements et rangements en mémoire. Également au niveau matériel, on voudra estimer la période d'horloge optimale, la dissipation de puissance, le pipelinage, etc. Au niveau des communications, le débit et la largeur des bus sont essentiels.

Le niveau transactionnel – de modèle de calcul DE – chez SystemC, est une excellente façon de modéliser des transactions caractérisées par un temps de départ, un temps d'arrivée et leur débit sur le bus [GLMS02]. L'outil VCC Cierto de Cadence [Schi99] offre d'étudier la performance par l'annotation des modèles architecturaux. La performance du logiciel est estimée par un modèle de processeur virtuel plutôt que par un ISS. Les multiples bus peuvent aussi être analysés par des statistiques amassées au cours de l'exécution.

1.4.4. Discussion sur les modèles

L'arrivée des langages OO offre plusieurs modèles de calcul à même un seul langage. Comme décrit plus haut, Ptolemy-II en fait la démonstration avec Java. SystemC apporte également plusieurs modèles dont KPN et SDF mais aussi un modèle transactionnel et un modèle RTL à événements discrets pour le niveau matériel (DE). Des composants d'un même SoC peuvent être plus facilement représentés par différents modèles pour mieux reproduire la réalité. Mais l'interaction de ces modèles distincts présente un dilemme pour le programmeur. Un des problèmes majeurs est celui de la représentation du temps à travers deux modèles qui se transmettent de l'information. Il est simple, en théorie, de constater qu'un modèle en englobe un autre, mais il en est tout autre dans la pratique. Des méthodes claires de raffinement ou d'interfaces doivent êtres établies dont une dans [GLMS02]. Autrement, on voudra formellement comparer deux modèles pour s'assurer que leurs comportements soient compatibles [LeSa98].

1.5. Langages et bibliothèques applicables à la conception des SoC

Un langage orienté système (*System-Level Design Language* ou SLDL) permet de définir un système embarqué, dans son ensemble selon plusieurs niveaux d'abstraction. Ceci rend la spécification homogène, i.e. un seul langage permet de décrire le système en entier : les algorithmes, les notions de temps et d'horloge et les concepts architecturaux, tels les microprocesseurs et les circuits spécialisés, les bus et leurs transactions, etc. Le même langage à tous ces niveaux facilite la co-simulation des modules matériel et logiciel. Toutefois, aucun langage existant ne peut réellement combler tous les domaines d'un système. L'industrie adopte de multiples solutions hétérogènes dont les plus populaires combinent le VHDL/Verilog avec le C/C++ [Klei02], [HFBA01].

Un langage système homogène devrait posséder les particularités orthogonales suivantes, telles que proposées dans [Sang97]. D'abord, il doit être indépendant de son implantation et sa description devrait être neutre pour décrire aussi bien le matériel que le logiciel. De plus, il doit être indépendant des outils avec lesquels il pourrait être rattaché. Il devrait supporter plusieurs modèles de calcul. Enfin, il devrait proposer une approche basée sur des raffinements progressifs.

Certaines particularités peuvent influencer le choix d'un langage pour la conception d'un SoC. Ces caractéristiques sont présentées dans [GVNG94]. Voici les plus importantes:

(1) La **concurrence** permet de simuler l'exécution parallèle de processus.

(2) La **hiérarchie**, structurelle ou comportementale, permet de décrire des systèmes de façon plus ou moins abstraite.

(3) Les **communications** entre les différents composants d'un système doivent être offertes.

(4) La **synchronisation** des processus concurrents d'un même système doit être également offerte.

(5) La **minutage** (*timing*) modélisant la réalité dans le système.

(6) Les **constructions de programmation**, permettant de décrire des algorithmes, par exemple des boucles, des tableaux de données, des procédures, etc.

1.5.1. C/C++/Java

À ce jour, les langages C et C++ ou encore Java ont souvent été utilisés pour modéliser et valider rapidement le comportement d'une spécification. Or, ces langages tels quels sont insuffisants pour représenter une spécification système pour une raison simple : ils ne supportent pas (ou que partiellement) les cinq premières caractéristiques essentielles énumérées ci-haut. Néanmoins, Java offre par ses bibliothèques de programmation des *threads* pour modéliser la concurrence. Les langages ou bibliothèques décrites dans les sections suivantes remédient à cette situation en implantant, grâce à l'orienté objet, une ou plusieurs de ces caractéristiques.

1.5.2. SystemC

SystemC [OSCI02a], [OSCI02b], [Swan01], [Duml02], supporté par OSCI, est une bibliothèque C++ orienté objet pour la modélisation de systèmes à plusieurs niveaux d'abstraction. Elle est de loin l'idée la plus populaire sur le marché actuellement. Une première version sortie en 1999 offre aux concepteurs la possibilité de modéliser le matériel en C++ en implantant les éléments requis pour la modélisation matérielle de base. Une deuxième version récente offre les concepts nécessaires à la modélisation de SoC à plusieurs niveaux d'abstraction, ce qui permet maintenant de spécifier un système à partir d'un niveau élevé puis en raffinant progressivement les spécifications vers le matériel ou le logiciel.

Le système est spécifié à l'aide d'objets tels des modules, des processus, des ports et des canaux. Des types de données abstraits ont également été définis. On fait communiquer les modules par les canaux grâce à des interfaces de programmation entièrement définissables. Des objets supplémentaires (canaux élémentaires) pour un meilleur support logiciel ont été ajoutés dont des files d'attentes (FIFO) et des sémaphores. On a

également ajouté des événements pour réveiller les processus internes, ainsi que des horloges de surveillance locales et globales (*watchdogs*) pour la gestion d'exceptions.

Les différents niveaux d'abstraction offerts sont tout d'abord les niveaux fonctionnels, avec ou sans notion de temps (respectivement nommés UTF et TF). Ces niveaux permettent la modélisation de multiples modèles de calcul, notamment KPN et SDF. Ensuite, un niveau transactionnel (BCA) permet de simuler des transactions (échanges entre composants) sur un bus abstrait par des communications bloquantes ou non. Des raffinements vers des modèles spécifiques, tels le modèle maître/esclave [Cyr01] sont possibles. Enfin, on effectue un raffinement supplémentaire vers le matériel au niveau horloge (PCA ou RTL) pour la communication point-à-point. On transforme les spécifications au niveau RTL pour une synthèse de SystemC ou une exportation vers des langages tels VHDL ou Verilog. Avec SystemC, il est possible de simuler des systèmes possédant plusieurs niveaux d'abstraction comme le démontre la figure 1.3.

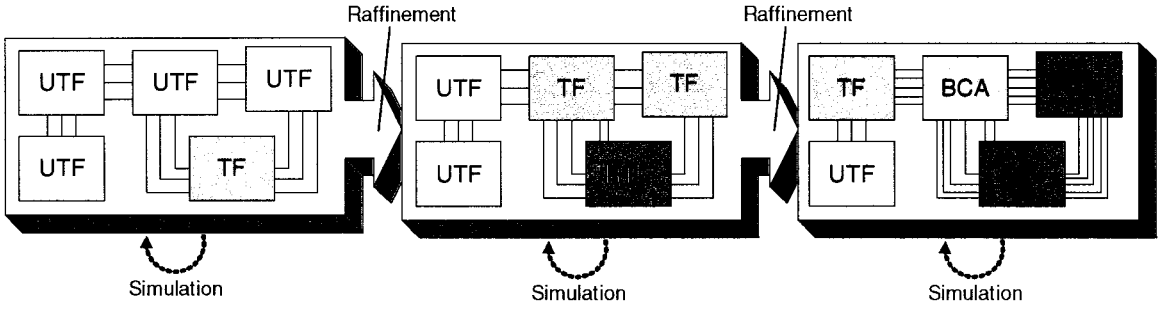


Figure 1.3 : Spécifications et simulations multi-niveaux avec SystemC

Différents modèles d'implantation sont proposés avec SystemC et tirés de [Schw02] et sont présentés à la figure 1.4. Ils mettent en opposition le niveau d'abstraction du modèle (ordonnée) au niveau de l'interface (abscisse). Le modèle SAM représente la plus haute abstraction possible pour un système complet. Les spécifications sont des fichiers exécutables UTF avec des protocoles abstraits. SPM ne reprend que des algorithmes abstraits et on s'en sert pour une évaluation grossière de la performance en y mesurant entre autre les communications. FM englobe l'ensemble des comportements d'un

système à haut niveau sans détails architecturaux. Avec TLM, on détermine le temps, avec ou sans horloge, des transactions sur le bus. TLM peut également être de niveau TF pour évaluer la fonctionnalité des modules sur des architectures. BSM couvre la synthèse comportementale et nécessite une interface PCA. BFM sert à la simulation PCA de module qui ne seront pas repris à la synthèse par exemple, un ISS. Enfin, RTL représente des spécifications de matériel classiques.

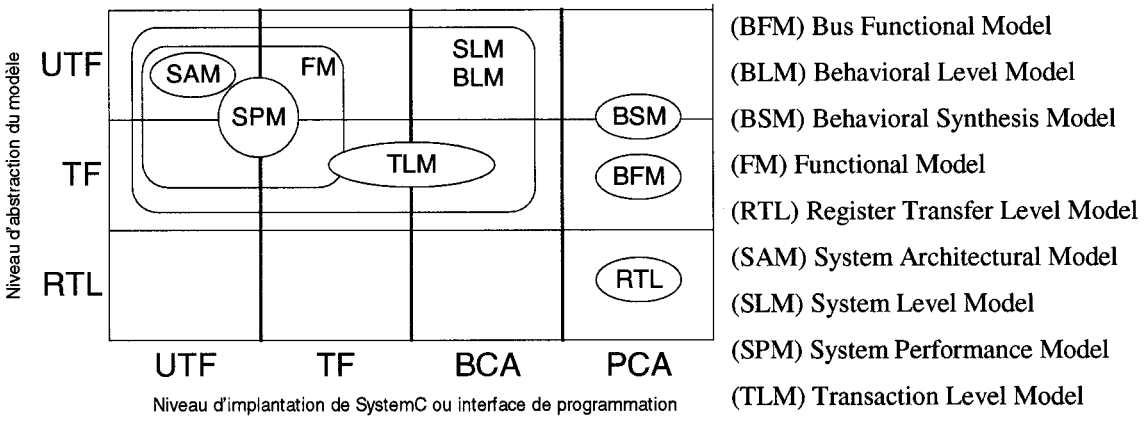


Figure 1.4 : Niveaux d'abstraction de SystemC

SystemC ne propose pas de méthodologie comme tel, il incarne plutôt un langage offrant diverses possibilités. Des outils industriels l'intègrent à leurs méthodologies comme nous le verrons à la section 1.6. SystemC offre cependant peu de support méthodologique pour le côté logiciel d'un SoC. Il prend pour acquis qu'un modèle de processeur connu sera implanté en SystemC ou qu'un ISS se branchera facilement à la simulation. Le langage n'offre pas non plus de normes ou de règles pour l'intégration de IP.

1.5.3. SpecC

SpecC [GZDG00], [DGG01] – proposé par Gajski et son équipe à Irvine – est basée sur une méthodologie complète qui débute par la spécification d'un système logiciel/matériel temporisé ou non au niveau fonctionnel. SpecC définit son propre langage qui se fonde sur les concepts C++ OO. Des canaux permettent de décrire des protocoles basés sur l'échange de données et d'événements. Le système demeure à haut niveau, mais le

minutage (*timing*) est exact ou encore défini sur un intervalle de temps. Les modules spécifiés peuvent être exécutés séquentiellement, concurremment, de façon pipelinée ou encore à partir de machines à états. SpecC approuve la réutilisation et l'intégration de IP et définit des règles à suivre pour l'utilisation d'adaptateurs de protocoles. De plus, SpecC propose une sémantique visuelle facile à comprendre et à utiliser.

SpecC offre tout ce qu'il y a de plus intéressant pour la spécification fonctionnelle: langage, compilateur, outil graphique pour la capture, simulation et analyse de performance. Les concepts logiciel ne sont pas négligés du tout, mais le matériel semble manquer de souffle, en particulier pour le raffinement vers le RTL, où d'autres outils doivent intervenir et se lier à SpecC. De plus, SpecC n'est pas tout à fait du C++ et le code source n'est pas ouvert. Ces éléments laissent le marché hésitant.

1.5.4. Cynlib

Cynlib [FDS00a], [FDS00b] par Forte Design Systems est une bibliothèque en C++ à code source ouvert pour la modélisation de matériel. Elle se lie à une méthodologie nommée Gigascale Hub (section 1.6.3). Cynlib permet une modélisation matérielle à haut niveau d'abstraction qui augmente de façon significative la performance d'une simulation comparativement aux simulateurs VHDL ou Verilog. Cynlib propose des constructions nécessaires au design matériel à l'aide de classes C++ supportant les concepts tels la concurrence des processus, l'assignation différée des signaux, les types de données orientés pour le matériel, la manipulation simplifiée sur les bits ainsi que les horloges. Lors de la construction du système, un design est partagé en modules (classes) dont chacun d'entre eux contient un à plusieurs processus. À l'exécution, ces processus sont activés soit par le changement d'un signal à l'entrée ou soit par le changement d'état de l'horloge. Il est alors possible de lire les nouvelles valeurs à l'entrée par les ports que possède le module, d'effectuer un traitement sur les données et de propager en sortie les nouvelles valeurs, tel un module combinatoire ou séquentiel.

La figure 1.5 montre le flot de conception de Cynlib. Ce flot intègre une série d'outils pour analyser la syntaxe de la spécification (Cyntax), pour compiler et déboguer une spécification (Cyngdb), ainsi que pour convertir la spécification en VHDL ou Verilog (Cynthesizer). D'autres outils permettent d'intégrer des modules en Verilog adaptés à Cynlib (Cyn++) ou encore des modules matériel tiers (Cynchronizer). Grâce à la technologie Gigascale Hub, une simulation hétérogène entre des modules Cynlib et des modules externes en VHDL ou Verilog peut être effectuée.

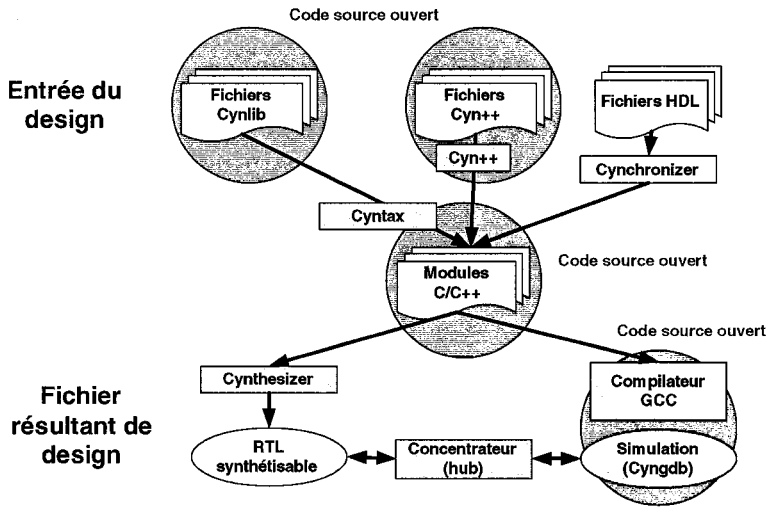


Figure 1.5 : Flot de conception Cynlib

Cynlib a maintenant rejoint le regroupement OSCI pour créer ESC: un nouveau produit représentant la convergence des bibliothèques Cynlib et SystemC [Sant01], ce qui implique que les designs en Cynlib sont maintenant exécutables avec SystemC. Forte affirme que ESC est plus rapide que le SystemC original pour les simulations RTL et que plusieurs restrictions de SystemC ont été retirées. ESC est présentement en développement et n'est malheureusement pas disponible au moment de la rédaction de ce mémoire.

1.5.5. OCAPI-xl

OCAPI-xl [VCPD99], [VSVE01] est une bibliothèque C++ pour le design de systèmes possédant des parties logicielles et matérielles. Le concepteur doit d'abord entrer les spécifications fonctionnelles d'un système en C/C++ sous forme de processus concurrents et temporisés. La spécification peut ensuite être vérifiée avec le simulateur *TIPSY* [VKS00]. Une fois l'analyse des performances effectuée, il faut transformer son code C++ en blocs de base et ce, en utilisant des objets OCAPI-xl décrivant le flot de contrôle d'un programme, par exemple des objets *boucles*, des objets *ifthen*, etc. Les communications doivent également être raffinées vers des objets de la bibliothèque : des variables partagées, des messages et des sémaphores. On assure ainsi le déterminisme des communications. Il s'agit d'un modèle complètement programmé avec des constructions OCAPI-xl et transformable par la suite vers des langages d'implantation. Le raffinement matériel peut se faire en utilisant un ensemble d'objets de niveau RTL, comme la machine à états, grâce aux concepts de classes et de surcharges d'opérateurs [VSB99].

Ce modèle proposé par l'IMEC intègre drastiquement des concepts orientés objets en englobant les moindres parties de code dans des objets de type OCAPI-xl. Cela garantit un calcul des performances plus efficace, mais demande beaucoup de temps au programmeur pour raffiner ses spécifications. Le modèle n'est pas lié à une architecture, ce qui cause sans doute un travail supplémentaire de vérification des communications une fois la spécification migrée sur sa plate-forme d'exécution.

1.5.6. UML

Unified Modeling Language [BRJ99] est un langage formel de modélisation général pour les systèmes logiciels qui offre à celui qui l'utilise une sémantique précise quant à l'analyse d'objets et au design. Il permet de construire, de spécifier et de visualiser graphiquement et à haut niveau des systèmes complexes de tout genre. Les structures, les classes et leurs instances ainsi que les relations sont autant de concepts clefs permettant de concevoir les bases d'un système. UML offre également les concepts OO d'héritage et

de polymorphisme. Des extensions de UML existent et sont plus propices au développement de SoC comme UML-RT [Seli99].

Comme la conception des SoC devient de plus en plus un problème de logiciel, il apparaît intéressant de voir comment on peut intégrer UML dans une méthodologie de conception des SoC. Dans [ArMa99], on propose d'utiliser le sous-groupe UML *Object Constraints Language* ou *OCL* et de l'intégrer dans une méthodologie de codesign basée sur les contraintes (*CBC* : *Constraints-Based Codesign*). La méthode est très formelle et s'apparente à ce qu'on retrouve dans Rosetta.

1.5.7. SDL

Specification and Description Language ou *SDL* est un langage formel orienté objet et spécialisé pour les télécommunications [BeHo89], [Dold01]. Son habilité à décrire graphiquement la structure, le comportement et les types de données d'un système sont ses forces majeures. La sémantique derrière les symboles SDL est bien définie et conçue pour les systèmes temps réel. Il est à noter que SDL peut communiquer avec des instances de plusieurs autres langages et que des outils existent pour transformer le code SDL en C/C++ exécutable. Les interfaces claires et biens structurées permettent une réutilisation et une vérification efficace des modules. Les interfaces font passer des signaux et des canaux de données séparément pour clarifier les designs complexes. Les processus, qui peuvent même être créés dynamiquement, sont réveillés grâce à ces signaux. Ils consomment et travaillent avec des données de types abstraits définis par l'usager. Enfin, des temporisateurs abstraits (matériel ou logiciel) bien définis et à usages multiples existent.

1.5.8. Rosetta et ALC

Le groupe Accellera supporte deux nouvelles approches basées sur des langages de spécification formels. Il s'agit de Rosetta [AKB00] qui est spécialisé pour les systèmes hétérogènes et offrant des constructions orientées sur les composants et les communications. Inspiré des langages de spécifications formels, tels Z [Word92] ou

Larch [GuHo93], Rosetta permet de capturer les spécifications d'un composant selon un domaine spécifique tel la machine à états finis ou infinis, l'orienté état, la logique pure ou encore le temps continu ou discret. Par ailleurs, ALC [Gero00] est un langage possédant une sémantique définie et une syntaxe reconnu (C++) et s'oriente sur la vérification, l'analyse et la synthèse. ALC s'inspire entre autre des idées de Ptolemy-II pour les modèles de calcul, de VSIA pour les types de données et de bibliothèques reconnues dont Cynlib et SpecC. ALC est présentement en développement.

1.6. Méthodologies de conception

Choisir une méthodologie à suivre avant de débuter tout design est sans aucun doute un atout. De façon générale, les méthodologies diffèrent entre elles par leurs aspects d'architecture, de vérification, de langage(s) de programmation. L'ingénieur choisit idéalement une méthodologie après en avoir analysé plusieurs, prenant celle qui répond à ses besoins. Plusieurs méthodologies n'intégrant pas les concepts de C++ ou de raffinement, plus anciennes celles-là, ont déjà fait leurs preuves. Les plus connues sont POLIS [BCGH97] utilisant le langage Esterel ou encore SpecSyn [GaRa94] utilisant SpecCharts [NVG92] et enfin COSMOS [VIJK94]. Ces méthodologies ne seront pas explicitées dans ce mémoire. Nous verrons plutôt comment il est possible d'intégrer l'orienté objet dans une méthodologie de codesign actualisée.

1.6.1. Méthodologies basées sur le C++

La venue des bibliothèques/langages orientés objet, sans méthodologie distincte, donne l'occasion à plusieurs groupes de recherche ou compagnie d'adapter ces bibliothèques à leurs propres besoins. C'est ce que propose Fayad et Khordoc dans [FaKh99] avec une méthodologie pour modéliser un décodeur vidéo MPEG2 (système orienté données). En partant d'un modèle en C++ *multi-thread* codé avec POSIX, ils déterminent les goulots d'étranglement de leur système qu'ils implanteront en matériel grâce à SystemC. Les modules matériels se lient à plusieurs interfaces implantant des protocoles de communication à différents niveaux d'abstraction. Les interfaces matérielles seront générées automatiquement grâce à Coware N2C [Cowa02].

Le groupe IMEC [IMEC02] propose également une méthodologie [VSVE01] basée sur son langage OCAPI-xl (bibliothèque C++). Une version fonctionnelle du système est spécifiée puis simulée de façon concurrente en testant plusieurs types de communications (mémoires partagées, messages, etc.) On raffine ensuite le système en orientant le travail vers des mécanismes de communication plutôt que vers une architecture prédéfinie. Ainsi, la spécification est indépendante de l'architecture utilisée. Enfin, la performance est analysée pour permettre à l'usager de créer un partitionnement. Les modules en OCAPI-xl sont transformés en C ou en VHDL automatiquement d'après les choix effectués.

1.6.2. VCC Cierto de Cadence

L'outil VCC Cierto [BaOb99], [Schi99] de Cadence présente une méthodologie de SoC complète utilisant des spécifications à haut niveau. On définit les spécifications en C++ supportées par une bibliothèque interne liée à l'outil. L'idée derrière VCC est d'intégrer des blocs prédéfinis (IP) pour augmenter la productivité puis de les simuler à un niveau fonctionnel. Avec VCC, on lie la fonction à l'architecture comme le démontre la figure 1.6. La spécification est liée à une plate-forme d'exécution abstraite et à un RTOS abstrait. Ces deux éléments sont sélectionnés à partir de bibliothèques mais amplement configurable par le concepteur. Les étoiles représentent les modes de communication. On effectue une simulation du système et on recueille des statistiques architecturales qui nous renseignent sur les changements à apporter au système (temps d'exécution détaillés, utilisation des bus, etc.) Des modèles architecturaux et de performance existent pour de nombreux processeurs et contrôleurs, pour des DSP, des bus, la mémoire et le cache.

La méthodologie offre l'environnement idéal pour l'exploration architecturale à haut niveau. Une grande variété de modèles de communication tirés des échanges logiciel/matériel, matériel/logiciel, logiciel/logiciel et matériel/matériel est offerte. La synthèse des spécifications matérielles est possible, à condition qu'elles soient codées selon le modèle de calcul CFSM [BCGH97]. Enfin, il est également offert d'exporter le

système au complet vers d'autres outils pour les étapes subséquentes. La méthodologie est par contre loin d'être intuitive et de nombreuses exceptions dans le flot de design doivent être connues. En outre, on n'en finit plus de fixer des paramètres avant de débuter les simulations. L'outil est complexe, trop complexe et même après une semaine de cours intensifs sur ses possibilités, on atteint à peine l'autonomie nécessaire pour bâtir et analyser un simple exemple producteur/consommateur.

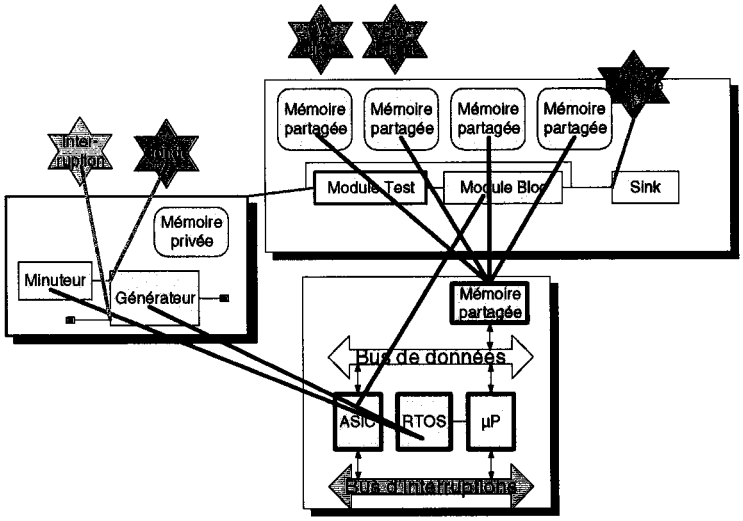


Figure 1.6 : Liaison de la fonction à l'architecture dans VCC Cierto

1.6.3. Gigascale Hub de Forte Design Systems

Forte Design Systems propose une méthodologie complète orientée plate-forme basée sur sa suite d'outil Cynlib : le Gigascale Hub [FDS02a]. Comme précédemment mentionné, il est prévu que Forte ajustera sa méthodologie à ESC au cours des prochains mois. Forte fait interagir les étapes de vérification fonctionnelle et de conception pour augmenter la productivité et ainsi diminuer le temps de commercialisation des systèmes. Cette interaction est illustrée à la figure 1.7.

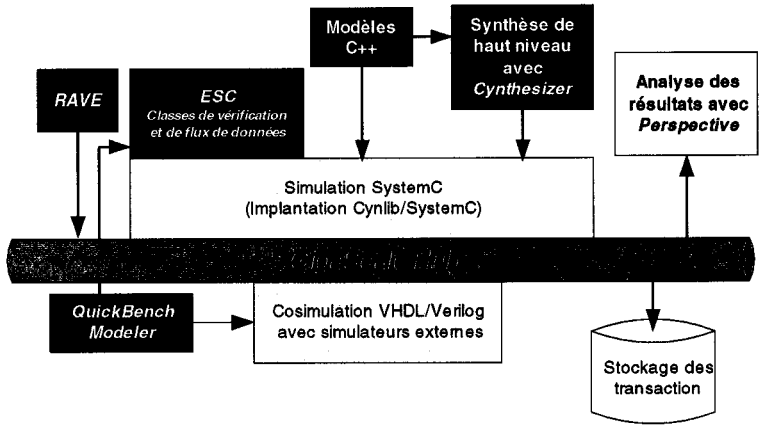


Figure 1.7 : La méthodologie Gigascale Hub de Forte Design Systems

Le Gigascale Hub est un bus qui permet d'intégrer des modules d'origines différentes. Premièrement, il est possible de brancher des modules Cynlib ou C++ de haut niveau. Ensuite, grâce à une génération d'interfaces pour communiquer sur un bus au niveau transactionnel, il est possible de mélanger des modules RTL à des modules de haut niveau (en Cynlib). Il est aussi possible de créer des bancs de tests (*testbench*) grâce à l'outil QuickBench en plusieurs langages : VHDL, Verilog, Cynlib (ESC) ou encore le langage de vérification fonctionnelle RAVE. Enfin, des modules de niveau RTL codés en différents langages (Cynlib, VHDL, Verilog) peuvent être connectés au bus. Pendant la simulation, il est possible d'analyser la performance du système conçu grâce au langage *Perspective*. Des résultats peuvent être recueillis pendant ou après la simulation pour ainsi valider entre autre la couverture fonctionnelle d'exécution, l'analyse temporelle et la vérification d'assertions tant pour les signaux à bas niveau que pour les transactions sur le bus à plus haut niveau [FDS02b].

1.6.4. CoCentric de Synopsys

CoCentric System Studio de Synopsys [Syno02] est un environnement de développement utilisant SystemC pour la spécification et la simulation système à tous les niveaux d'abstraction. Les spécifications peuvent être visualisées de plusieurs façons dont graphiquement, symboliquement, code source des composants et interfaces. Des éditeurs concrets permettent de créer différents modèles basés sur une banque d'algorithmes

utiles. Un environnement de vérification supporte la création de *testbenchs* orientés vers les données. Un co-simulateur puissant fournit des utilitaires pour espionner le contenu des bus, pour tracer des diagrammes temporels, pour guetter l'ordonnancement, pour analyser des statistiques recueillies graphiquement, etc. Il peut également se lier à des simulateurs externes pour le matériel, tels ModelSim. Côté synthèse, CoCentric peut générer du code SystemC synthétisable qu'on peut utiliser avec d'autres outils de Synopsys pour la création de modèles physiques ou de FPGA.

1.6.5. Méthodologie basée sur Renoir 2000, Seamless CVE & C-Bridge

Les compagnies ont parfois tendance à baser leurs méthodologies sur des outils industriels, n'ayant pas de temps ou d'argent à consacrer à la recherche d'une méthodologie basée sur des concepts. Il est par conséquent possible d'utiliser une série d'outils complexes pour développer un système embarqué. À titre d'exemple, un travail d'intégration au laboratoire nous a permis de développer une application simplifiée de reconnaissance de patron (partie de l'encodage vidéo MPEG) nommée *BlockMatcher*. La méthodologie utilisée mettait en œuvre l'outil de co-simulation Seamless CVE [Ment00] et l'outil de conception de matériel Renoir 2000. Le *BlockMatcher* sert maintenant de tutoriel [BeFi01] pour apprendre à travailler avec l'outil Seamless CVE. Le *BlockMatcher* a d'abord été programmé en C/C++ pour en vérifier la fonctionnalité. Ensuite, un partitionnement manuel a été effectué et les spécifications ont été utilisée pour recoder (complètement avec l'aide de Renoir 2000) le *BlockMatcher* en matériel (VHDL). L'application a été intégrée à la plate-forme *BasicARM* [Hene01] puis co-simulée avec Seamless CVE.

Le temps consacré au développement du *BlockMatcher* nous a permis de comprendre le travail d'un ingénieur de l'industrie. Le problème avec cette méthodologie est qu'on ne « ressent » le système que lorsqu'on intègre les modules sur la plate-forme architecturale. Malheureusement, cette intégration tardive diminue énormément l'espace de recherche au niveau architectural. Cette absence aura sans doute influencé le design dès le plus haut niveau d'abstraction. Une extension à ce tutoriel inclura bientôt l'outil C-Bridge

[Ment02b] qui permet de simuler des blocs de matériel ou de logiciel en C/C++ conjointement avec l'outil Seamless CVE.

1.7. Rétrospective

Les langages et méthodologies présentés peuvent se comparer entre eux quant aux aspects de spécifications qu'ils couvrent. En effet, tout concepteur devra en principe évaluer les langages qui s'offrent à lui avant de sélectionner celui qui convient le mieux au système à spécifier. Cette comparaison est effectuée à la figure 1.8. On y présente les principaux langages décrits à la section 1.5, en plus de VHDL, Verilog et SystemVerilog. Les cas de SystemC sont déjà présentés à la figure 1.4 et ne sont pas repris. La méthodologie VCC est également représentée. Nous montrons également où se situe la bibliothèque Syslib (Syslib^{FL} et Syslib^{BL}) que nous allons présenter au cours du prochain chapitre.

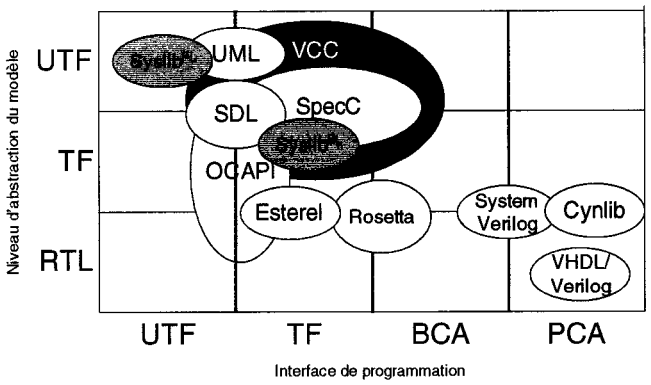


Figure 1.8 : Comparaison des différents langages, bibliothèques et méthodologies[2]

Plusieurs langages ou bibliothèques couvrent les aspects fonctionnels de spécification du modèle, c'est le cas pour UML, SDL, SpecC et OCAPI-xl. Ces bibliothèques offrent toutes une interface de programmation de haut niveau. Certaines proposent une interface orientée sur les échanges de bus, comme SpecC ou Rosetta. Par ailleurs, Esterel, Rosetta

[2] Note : UTF : *Untimed Functional*, TF : *Timed Functional*, BCA : *Bus-Cycle Accurate*, PCA : *Pin-Cycle Accurate*, RTL : *Register Transfer Level.*

et SystemVerilog demandent de fournir plus de détails pour décrire les modèles. Ce dernier est le seul à tenter de combiner des aspects de bus au matériel RTL. Enfin, VHDL, Verilog et Cynlib couvrent les aspects de modélisation de matériel. De plus, la méthodologie VCC permet de spécifier des modèles avec ou sans notion de temps en utilisant une interface de programmation abstraite (communication sans temps à des transactions sur bus).

CHAPITRE 2
Méthodologie de développement

Il importe pour un concepteur de SoC d'avoir une vision de conception utilisant une méthodologie de codesign concrète. Ce choix permet de modéliser des systèmes rigoureusement, en empruntant une voie prédéfinie qui oriente les idées des programmeurs. Ainsi, une méthodologie détaillée intégrant la bibliothèque Syslib sera explicitée.

Ce chapitre présente également la bibliothèque Syslib et ses fonctionnalités de base à l'aide d'un court exemple pour que le lecteur se familiarise avec la syntaxe, l'interface et les concepts de Syslib. Quelques détails de l'implantation de la bibliothèque sont finalement présentés.

2.1. Méthodologie Syslib

Une analyse soignée des requis et des avoirs a mené à l'établissement d'une méthodologie de codesign détaillée et solide : la méthodologie Syslib, telle qu'illustrée à la figure 2.1. La méthodologie Syslib mérite d'être explicitée en détail et c'est ce que nous ferons dans cette section. Elle tente de s'intégrer aux méthodes actuelles du marché en utilisant des bibliothèques et des outils professionnels concrets.

Au départ, une spécification est bien souvent décrite à l'aide de mots et de phrases qu'on implantera avec un langage de haut niveau pour valider son comportement, ses algorithmes. Le C/C++ est sans doute le plus populaire, mais il demeure un langage parmi d'autres. C'est une fois cette spécification implantée qu'intervient la méthodologie Syslib. D'abord, on doit rendre la spécification modulaire en la raffinant d'éléments fondamentaux nécessaires à la description de systèmes. Nous utilisons des concepts de modules, ports, canaux, événements et variables pour spécifier la structure d'un système.

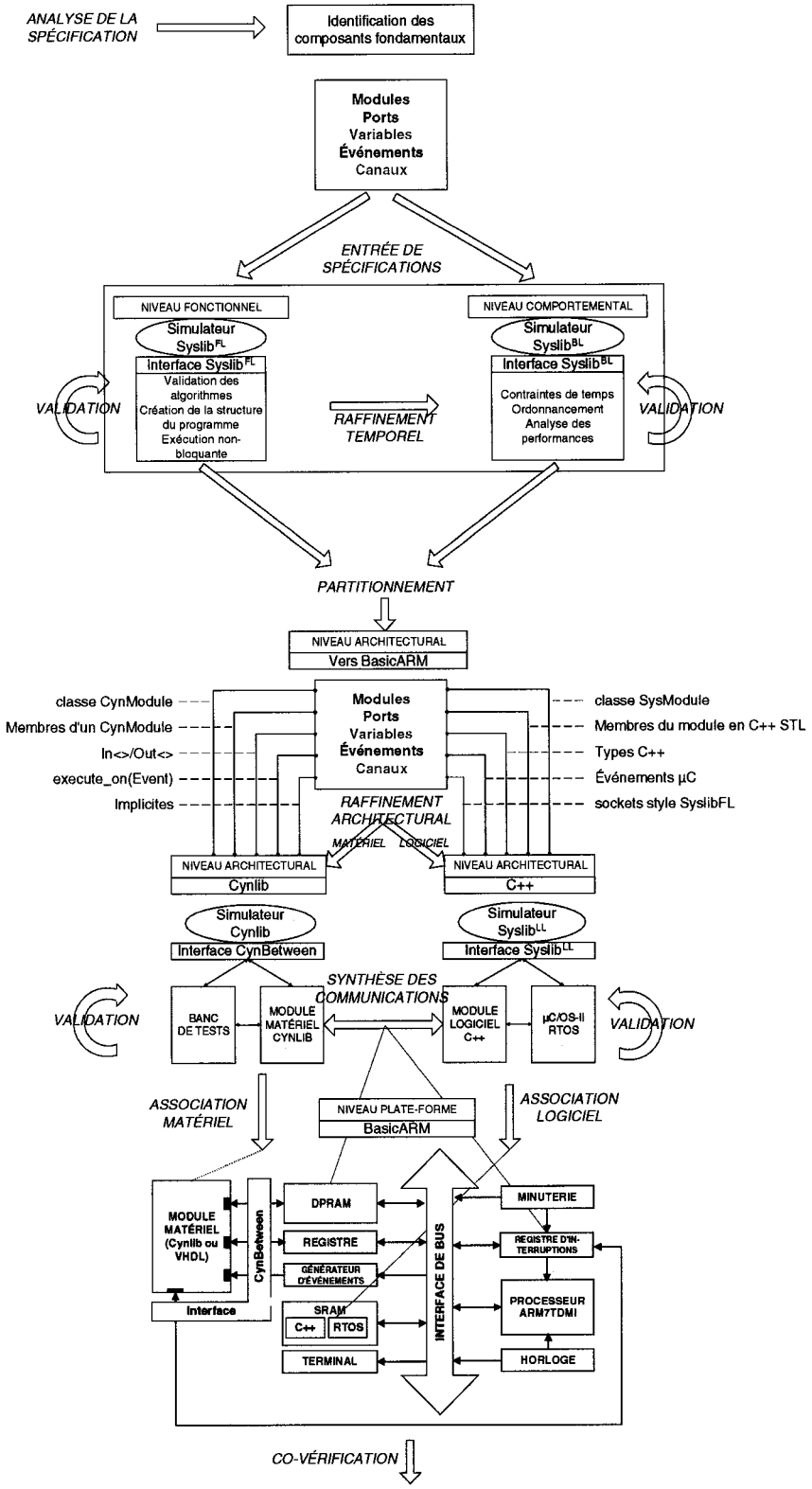


Figure 2.1 : Méthodologie de codesign Syslib

On crée alors une première spécification à l'aide de Syslib^{FL}. Il s'agit du **niveau fonctionnel** (FL), un niveau qui contraint les spécifications à n'utiliser aucune notion de temps. La spécification raffinée est ainsi validée à l'aide de simulations multiples. Un **raffinement temporel** fait passer la spécification au **niveau comportemental** (BL avec Syslib^{BL}) où une analyse plus poussée de l'ordonnancement sera effectuée. Il s'agit d'un niveau possédant des notions de temps « fonctionnelles » avec des temporisateurs, des contraintes de temps, etc. Là aussi, des simulations permettent d'évaluer le modèle. Chacune de ces bibliothèques est indépendante, ce qui fait qu'on développe avec l'une ou l'autre, ou un mélange des deux (FL est supporté par BL). D'autre part, la syntaxe menant de l'une à l'autre est la même, et donc aucune transformation syntaxique n'est requise en passant de FL à BL.

Une fois les validations terminées, un **partitionnement** manuel est effectué pour lier la spécification à l'architecture BasicARM [Hene01] sur laquelle les spécifications seront insérées. Chacun des concepts fondamentaux constituant la spécification sera implanté, soit en matériel en raffinant la spécification vers Cynlib, soit en logiciel en intégrant un RTOS aux modules.

Pour l'implantation matérielle en Cynlib, on propose d'abord un outil d'analyse syntaxique qui transforme automatiquement la spécification de Syslib vers Cynlib. Les modules matériels devront être validés à l'aide de bancs de tests en Cynlib ou en VHDL. L'hétérogénéité de la simulation est résolue par l'utilisation de PLI [HFBA01]. L'interface *CynBetween* est compatible à celles de Syslib et donne accès à des méthodes qui implantent les communications intermodules. Du côté logiciel, la spécification est raffinée en C/C++ (pour limiter la taille du code) et liée par une interface Syslib^{LL} au RTOS µC/OS-II [Labr99]. Cela donne accès à des constructions logicielles telles les sémaphores, les FIFO, les variables partagées, etc.

La simulation finale passe d'abord par une étape d'**association** (*binding*) des modules matériel à des éléments de plate-forme. Une compilation logicielle s'effectue et les modules logiciel (et le RTOS) sont placés en mémoire sur la plate-forme. Une **synthèse des communications** (pour le moment manuelle) doit être effectuée pour permettre aux informations de circuler entre le logiciel et le matériel à travers des éléments physiques de la plate-forme : des mémoires partagées (DPRAM), des registres, un générateur d'événements et un gestionnaire d'interruptions. La plate-forme sera configurable dans le cadre d'un autre projet [Bert03]. Enfin, le système passe à l'étape de **co-vérification** grâce à l'outil Seamless CVE de Mentor Graphics. Les modules matériels de la plate-forme (VHDL et Cynlib) sont synthétisables, ce qui nous permet de croire en la fabrication de prototypes et éventuellement à la réalisation de SoC réels. La méthodologie a été adoptée, mais des changements y ont été apportés en cours de route et seront présentés en guise de conclusion.

Ce projet de maîtrise se concentre sur le développement du niveau fonctionnel de la bibliothèque Syslib^{FL}. Une démonstration sera également effectuée au niveau du raffinement vers Cynlib. Les aspects Syslib^{BL} et Syslib^{LL} seront brièvement explicités, mais ne seront pas implantés et feront partis des travaux futurs.

2.2. Analyse préliminaire de Syslib

Lorsqu'on se rapporte à la méthodologie, Syslib possède trois niveaux distincts et ces niveaux sont expliqués ci-dessous. Cette section représente donc la vision, les objectifs de la bibliothèque Syslib en discutant davantage des niveaux de la figure 2.1.

2.2.1. Niveau fonctionnel ou Syslib^{FL}

Le niveau d'abstraction le plus élevé de la méthodologie est appelé « niveau fonctionnel ». À ce niveau, un concepteur valide l'idée générale ou l'algorithme rattaché à l'application à implanter sur SoC, sans se soucier des détails architecturaux ou du partitionnement. À cette étape, aucun détail sur la temporisation ou la synchronisation

n'est employé. On ne considérera qu'un ordre relatif d'exécution (précédance), sans plus (par exemple : le module *i* s'exécute avant le module *j*).

2.2.2. Niveau comportemental ou Syslib^{BL}

Ici, le programmeur enrichit son design au niveau temporel en faisant apparaître des détails d'implantation. Il s'agit d'un processus de raffinement. Syslib^{BL} est donc un ensemble supplémentaire d'éléments permettant la temporisation des modules, mais toujours à un niveau d'abstraction élevé. Il est possible de mettre les modules en attente active pour une période de temps donnée ou alors de définir des contraintes de temps strictes qui influencent l'ordonnancement des modules. La préemption des modules devrait permettre de mieux modéliser le comportement d'un système. Quelques statistiques d'exécution peuvent être recueillies pour prendre des décisions quant au partitionnement logiciel/matériel.

2.2.3. Niveau architectural ou Syslib^{LL}

Une fois les étapes fonctionnelles et comportementales terminées, il importe d'implanter les modules dans leurs domaines effectifs avec les langages qui s'imposent. Les modules partitionnés en logiciel ont besoin d'un RTOS pour supporter la gestion des *threads*. Syslib^{LL} représente donc l'API nécessaire à cette intégration du RTOS à l'application. Ce niveau doit supporter la gestion des interruptions et la génération d'événements pour que le microprocesseur puisse communiquer avec le monde extérieur. L'utilisation du micronoyau µC/OS-II répond à ces besoins. Il est simple et efficace et possède toutes les caractéristiques qu'un concepteur pourrait désirer. L'interface de Syslib^{LL} doit se conformer à celles des deux niveaux précédents pour minimiser le travail de raffinement.

2.3. Décisions d'implantation

Dans cette section, nous verrons les principales décisions d'implantation qui ont permis d'aboutir à Syslib^{FL} tel qu'il existe aujourd'hui. Plusieurs concepts de Cynlib ont été étudiés afin de faciliter le raffinement matériel vers cette bibliothèque.

2.3.1. Reprise du code de Cynlib

Au tout début, nous avions considéré d'intégrer le code de Syslib^{FL} à même la bibliothèque Cynlib. Après une analyse approfondie, cette idée a été rejetée puisque Cynlib est gigantesque et ne vient avec aucune documentation. Les modifications auraient été plus difficiles à implanter sur les futures versions de Cynlib. De plus, le simulateur matériel de Cynlib est trop lent et n'est pas adapté au logiciel. Syslib^{FL} a donc été programmée à partir de zéro.

2.3.2. Orientation générale

Syslib^{FL} offre, via un API similaire à celui de Cynlib, un nombre de services supportés par la *Standard Template Library* [DeDe98]. Les services originels proposés qui ont été conservés sont les suivants:

- Des *threads* pour ordonnancer les modules.
- Des *ports-sockets* pour permettre la communication entre les modules.
- La gestion d'événements par des fonctions de rappel.
- Aucune notion de temps.

2.3.3. Modules

Le module est un contexte d'exécution géré par un *thread*. Après analyse, nous avons décidé d'utiliser un ordonnancement dynamique de type FIFO sans préemption. Une étude plus approfondie sur Cynlib a montré qu'elle utilise la bibliothèque de *threads* POSIX [Gray98] pour obtenir des contextes d'exécution. Or, l'ordonnancement des modules est effectué par le simulateur interne de Cynlib. La fonctionnalité apportée à Cynlib par POSIX aurait pu être obtenue en utilisant les fonctions classiques C *longjump* et *setjump* [Micr02], ce que nous faisons avec Syslib^{FL}. Les modules de Syslib contiennent plusieurs unités d'exécution que l'on nomme aussi fonctions de rappel (ces concepts seront expliqués en détail plus loin).

2.3.4. Ports et canaux

On trouve dans la bibliothèque Cynlib que les ports rattachés aux modules sont connectés à d'autres ports à l'aide de *sockets* simplifiés et atrophiés de leurs fonctionnalités. Cette notion de *port-socket* sera nommée **capsule**. Ces rudiments sont affichés à la figure 2.2. On passe en paramètre au constructeur du module Cynlib des capsules qui servent de liens entre les ports appartenant au module et les ports d'un module externe. Le port et la capsule dans Cynlib sont représentés par les mêmes classes, soient *In* et *Out*. Il n'y a pas de files d'attente ni de consommation des données, mais la valeur du port est persistante.

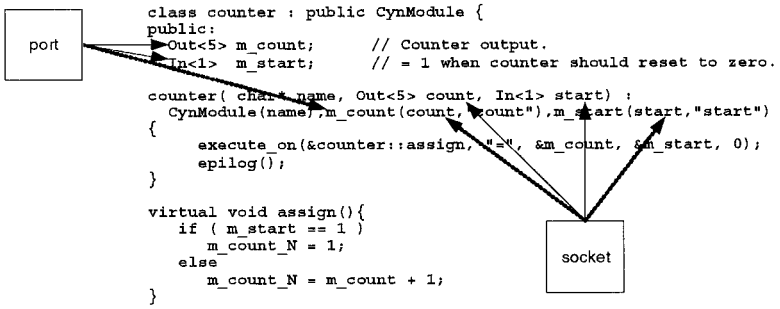


Figure 2.2 : Notions de ports et de *sockets* dans Cynlib

Nous utilisons une structure semblable à Cynlib pour simplifier le raffinement de la spécification vers le matériel. Comme l'approche Syslib est orientée objet, les ports et les capsules (qui deviendront des canaux), sont des objets distincts. Beaucoup de concurrents utilisent les RPC pour implanter ce type de communication dont SpecSyn [GaRa94] et Coware [Cowa02], mais nous rejetons ce paradigme de fonctionnement puisqu'il est basé sur le modèle maître/esclave qui est trop limitatif pour un système : on ne veut pas se restreindre à ce seul type de comportement pour concevoir des systèmes.

Pour les échanges, les données sont transmises entre les modules comme dans Cynlib, à l'aide de capsules pour en protéger la consistance. Nous appelons ces capsules des canaux de communication. Plusieurs concurrents incorporent la notion de protocole aux canaux, comme le *handshaking*. Avec Syslib^{FL}, les protocoles ne sont pas implantés

explicitement sur un canal comme avec SpecC. Ils sont plutôt représentés par le rassemblement de plusieurs canaux de données et d'événements. Pour créer un protocole spécifique, on doit programmer un module supplémentaire qui encapsule le comportement de ce protocole.

2.3.5. Communications, événements et données

Après analyse, nous avons décidé de ne pas implanter de communications bloquantes, afin de réduire les interblocages (*deadlocks*). On pourra ainsi valider plus rapidement l'algorithme du système et on optera dès le niveau fonctionnel pour une approche orientée vers le matériel.

Nous utilisons les événements pour réveiller les modules endormis. Les modules sont réveillés sur la notification de l'événement, soit en fin de *thread* lorsque l'exécution d'un module est terminée ou en milieu de *thread*, par exemple après la vérification d'une condition. Nous implantons les données au moyen de types de données abstraits. Nous implantons ces types à l'aide de redéfinitions de types (*typedef*). Les données sont consommées à partir du canal, c'est-à-dire qu'on retire la donnée de la file lorsqu'elle est lue. L'interface simule un comportement de persistance (la dernière donnée lue est persistante dans le port).

2.3.6. Assemblage du tout

Un schéma représentant la vision finale de la structure incluant modules, ports, canaux, événements et données est présenté à la figure 2.3. Dans cet exemple, le module 1 possède un port de données (v_out), associé à un canal (v) d'une profondeur de 4 éléments ainsi qu'un port d'événements (e_out) associé à un canal (e). Le module 2 est rattaché, par ses ports, aux canaux v et e.

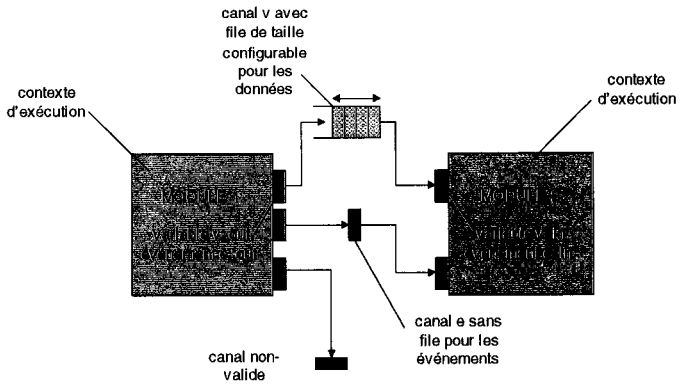


Figure 2.3 : Les concepts structurels de Syslib

2.4. Exemple de système

Mettons tous ces concepts à l'œuvre à l'aide d'un court exemple qui décrit bien la base des constructions de Syslib^{FL}. Notons que pour des fins de simplicité, nous utiliserons le terme **Syslib** pour représenter la bibliothèque au niveau fonctionnel. Cet exemple de système se nomme *PacketRouter* et il permettra au lecteur de se familiariser avec Syslib. On présentera également une partie du code rattachée à l'implantation de cet exemple. Le reste du code se retrouve en Annexe F.

2.4.1. Le *PacketRouter*

Le *PacketRouter* ou routeur de paquets est un exemple simple de type producteur/consommateur. La figure 2.4 schématise ce routeur de paquets. Le schéma utilise un gabarit de design simple et spécialement développé pour Syslib. Une légende présentée au tableau 2.1 décrit les symboles utilisés dans ce gabarit.

L'exemple de la figure 2.4 présente le nom de certains canaux ou ports. Pour éviter d'alourdir la figure, seuls les ports et canaux mentionnés dans ce texte sont indiqués. De plus, pour éviter toute confusion, les noms des canaux débutent par *c* et les noms des ports débutent par *p*. Trois modules composent cet exemple : un routeur (*PacketRouter*) fait circuler des paquets (les paquets sont représentés par des entiers pour fins de simplicité), un réseau (*Network*) et un pilote (*Driver*).

Tableau 2.1 : Légende du gabarit de design Syslib

Module		Communication		Unité d'exécution	
nom	de premier niveau (MPN)		port	sendData()	fonction de rappel (et son prototype)
			canal d'événements		
nom	hiérarchique (MH)		canal de données (avec profondeur)		réactive
		cFull <SysEvent>	objet<type>		perpétuelle

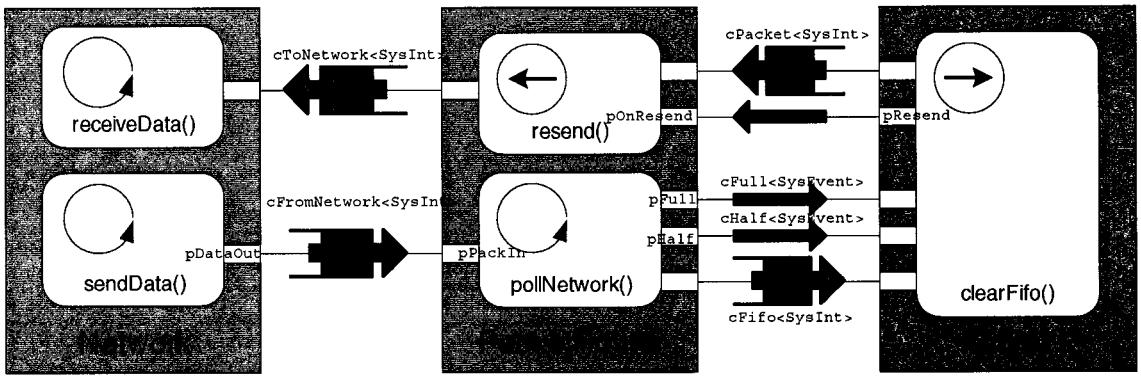


Figure 2.4 : Exemple du *PacketRouter*

Ces modules sont interconnectés par des canaux d'événements et de données. Les canaux de données sont des files d'attentes et leur profondeur est indiqué sur le symbole. Le module *PacketRouter* contient deux unités d'exécution: *resend* et *pollNetwork*. L'unité *pollNetwork* est perpétuelle, alors que *resend* est réactive. Les unités perpétuelles (définies en détail à la section suivante) peuvent représenter des comportements tels l'interrogation (*polling*) puisqu'elles ordonnancent automatiquement les modules pour exécution. *PacketRouter* est la clef de cet exemple, il peut constituer le module à évaluer pour un futur partitionnement. Il est relié à deux autres modules qui peuvent être vus comme des bancs de tests. *Network* est constitué de deux unités perpétuelles; *sendData* qui envoie des paquets dans le système et *receiveData* qui en récupère quelques-uns. Enfin, le module *Driver* répond à son environnement grâce à l'unité d'exécution réactive

clearFifo. Dans le cas où *Driver* ne serait pas assez rapide, il renvoie le dernier paquet reçu vers le routeur : ce dernier le fera transiter vers le réseau pour qu'il soit réémis.

La simulation débute avec le module *Network* qui s'ordonnance automatiquement parce qu'il contient des unités d'exécution perpétuelles. L'unité *sendData* génère un nombre aléatoire d'entiers (de type *SysInt*) qu'elle envoie dans le canal *cFromNetwork* par son port *pDataOut*. Parce qu'il contient des unités perpétuelles, le module se réordonnance pour exécution. Puis, c'est le module *PacketRouter* qui se réveille pour exécuter son unité perpétuelle *pollNetwork*. Il lit les données de son port d'entrée *pPackIn* et les fait transiter vers le canal *cFifo*. Si le canal est rempli à plus de la moitié, une notification sera effectuée sur le port *pHalf*. Dans le cas où le canal serait plein, c'est le port *pFull* qui sera notifié. Le routeur termine son exécution. Le module *Driver* qui est sensible aux événements circulant sur les canaux *cHalf* et *cFull* sera ordonnancé. À son tour, il consommera les données du canal *cFifo*. Dans les cas où il aurait été réveillé par un événement indiquant un canal plein, il renverra le dernier paquet reçu vers le canal *cPacket* et notifiera le port *pResend*. À son exécution, l'unité d'exécution *resend* du *PacketRouter* lira ce paquet et l'enverra vers le canal *cToNetwork*. Ce paquet sera lu par le réseau lors de l'ordonnancement de *Network*.

2.4.2. Fichier de structure des modules

Pour mieux comprendre cet exemple, voyons le code rattaché au module *PacketRouter*. Tout module est séparé en deux fichiers : l'en-tête (*PacketRouter.h*) qui contient la structure du module, puis la source (*PacketRouter.cpp*) qui contient l'implantation des unités d'exécution. D'abord, voyons la structure du module définie dans le fichier *PacketRouter.h* présenté à la figure 2.5.

La première étape est d'inclure le fichier de bibliothèque *Syslib.h* au module. Chaque nouveau module est dérivé (par héritage) de la classe principale *SysModule*, qui contient les services de base pour la création de modules (ligne 6). Le module possède un seul constructeur (ligne 10). Les ports servent soit à l'entrée d'information (*SysInPort*) ou à la

sortie d'information (*SysOutPort*). Les ports d'entrée et de sortie combinés n'existent pas (*SysInOutPort*) mais pourraient éventuellement être implantés. Le type du port (*SysInt* ou *SysEvent*) est spécifié à même sa définition, à l'intérieur des crochets (représentant les *templates* C++). Le constructeur et les ports sont déclarés publiquement pour être visibles de l'extérieur. La partie privée (mot clef *private*) contient l'en-tête des unités d'exécution. Les autres membres ou méthodes internes devraient être privés pour respecter la sémantique d'un système qui oblige à communiquer des informations d'un module à l'autre que par l'utilisation des ports.

```
 1 #ifndef PacketRouter_H
 2 #define PacketRouter_H
 3
 4 #include "Syslib.h"
 5
 6 class PacketRouter : public SysModule {
 7
 8 public:
 9    // déclaration du constructeur
10    PacketRouter();
11
12    // déclaration des ports
13    SysInPort<SysInt>     pFromNetwork;
14    SysOutPort<SysInt>    pToNetwork;
15    SysOutPort<SysInt>    pFifo;
16    SysInPort<SysInt>     pPacket;
17    SysInPort<SysEvent>   pOnResend;
18    SysOutPort<SysEvent> pFull;
19    SysOutPort<SysEvent> pHalf;
20
21 private :
22    // déclaration des fonctions de rappel
23    void pollNetwork();
24    void resend();
25
26    // déclarations personnelles ici (objets, etc)
27 };
28 #endif //PacketRouter_H
```

Figure 2.5 : Fichier de structure (.h) du module *PacketRouter*

2.4.3. Fichier d'implantation des modules

Nous voyons ici l'implantation du module *PacketRouter* qui se fait dans le fichier source (*PacketRouter.cpp*). La première partie de l'implantation consiste à inclure la structure du module fournie dans le fichier *PacketRouter.h* à la figure 2.6.

```
1  #include "PacketRouter.h"
2
3  PacketRouter::PacketRouter()
4  {
5     SysAddBehaviour(&PacketRouter::resend, &pOnResend, 0);
6     SysAddBehaviour(&PacketRouter::pollNetwork, 0);
7  }
8
9  void PacketRouter::pollNetwork()
10 {
11    int pushedvalues = 0;
12    bool notified = false;
13
14    while (1) // exécution jusqu'au vidage du canal d'entrée
15    {
16       if (pFromNetwork.read() == SYSCHANNEL_EMPTY)
17          return; // plus de valeur à consommer, fin de l'exécution
18       else
19          pFifo = pFromNetwork;
20
21       if (pFifo.write() == SYSCHANNEL_FULL)
22       {
23          pFull.notify();      // notification au Driver: le fifo est plein
24          return; // fin de l'exécution
25       }
26       else
27       {
28          if ( ++pushedvalues > MY_CHANNEL_SIZE/2)
29          {
30             // fifo remplit à plus de la moitié, notifier Driver
31             if ( notified == false )
32             {
33                pHalf.notify() != SYSCHANNEL_ALREADYACTIVE);
34                notified = true;
35             }
36          }
37       }
38    }
39 }
40
41 void PacketRouter::resend()
42 {
43    if (pOnResend.peek()) // regarder si le port a été notifié.
44    {
45       // consommer l'événement
46       pOnResend.consume();
47
48       // faire transiter le paquet
49       pPacket.read();
50       pToNetwork = pPacket;
51       pToNetwork.write();
52    }
53 }
```

Figure 2.6 : Fichier d'implantation (.cpp) pour le module *PacketRouter*

Le constructeur (lignes 3-7) se doit de déclarer les règles d'activation du module: il s'agit de créer des unités d'exécution (des *SysBehaviours*) en associant des ports d'événement à une fonction de rappel du module. Par exemple à la ligne 5, on associe la fonction de

rappel *PacketRouter::resend*, une méthode de *PacketRouter*, au port d'événement *pOnResend* (entrée de *PacketRouter*). Notez qu'il est possible d'ajouter autant de ports d'événement que voulu, puis de terminer avec le paramètre 0 (NULL). À la ligne 6, on crée une unité d'exécution perpétuelle en ajoutant un *SysBehaviour* sans fournir de ports d'événement en paramètre. Il est également permis de lier le même événement à plusieurs unités d'exécution différentes.

Il apparaît important de mentionner qu'une fonction de rappel n'est pas bloquante et par conséquent, qu'elle doit terminer son exécution (par le mot clef *return*) pour céder l'exécution à un autre module (comme c'est le cas à la ligne 24). La section 3.1.1 donne plus d'explications sur ce concept. L'algorithme de la fonction de rappel *pollNetwork* est simple et consiste à lire les valeurs du port *pFromNetwork* (ligne 16) une à une grâce à la méthode *read*, à les copier vers le port de sortie *pFifo* (ligne 19) grâce à la surcharge de l'opérateur =, puis à les propager dans le canal (ligne 21) avec la méthode *write*. La valeur du port est accessible au moyen de la méthode *value*. Advenant que le canal relié à *pFifo* soit rempli à plus de la moitié, *PacketRouter* notifie un événement au module *Driver* (par les lignes 23 ou 33). La fonction de rappel *resend* est exécutée dans les cas où une notification arrive sur le port d'événement *pOnResend*. Il est possible de lire l'état du port d'événement, sans consommer l'événement par la méthode *peek* (ligne 43). On désactive l'événement en le consommant avec *consume* (ligne 46). Notez le fait que si un événement n'est pas consommé, son état demeure actif et aucune nouvelle notification ne sera possible. Dans cet exemple, il apparaît inutile de vérifier l'état de l'événement avec la méthode *peek*, puisque l'on sait que si la fonction de rappel est exécutée, c'est par une notification sur le seul port associé à cette unité, le port *pOnResend*. Toutefois, il est possible que plusieurs ports d'événement soient liés à une même fonction de rappel. Il est ainsi possible de connaître l'événement qui a réveillé cette unité et ainsi exécuter les bonnes parties de codes. Un exemple de ceci est présenté en Annexe F dans le fichier *Driver.cpp*.

2.4.4. Fichier global du système

Il s'agit maintenant d'instancier tous les modules décrits et de les connecter par des canaux appropriées. Voici, à la figure 2.7, une partie du fichier principal qui fait ce travail.

```
 1 #include "Network.h"
 2 #include "PacketRouter.h"
 3 #include "Driver.h"
 4 #include "SysOS.h"
 5
 6 int main(void)
 7 {
 8    // instance des modules
 9    Network mnetwork;
10    PacketRouter mpacketrouter;
11    Driver mdriver;
12
13    // instance des canaux entre Network et PacketRouter
14    SysChannel<SysInt> cToNetwork;
15    SysChannel<SysInt> cFromNetwork;
16    cToNetwork.setFIFODepth(32);
17    cFromNetwork.setFIFODepth(32);
18
19    // instance des canaux entre PacketRouter et Driver
20    SysChannel<SysInt> cPacket;
21    SysChannel<SysInt> cFifo;
22    SysChannel<SysEvent> cFull;
23    SysChannel<SysEvent> cHalf;
24    SysChannel<SysEvent> cResend;
25    cFifo.setFIFODepth(32);
26
27    // association des ports et des canaux
28    cFull.bind((SysModule*)&mpacketrouter, &mpacketrouter.pFull,
29                (SysModule*)&mdriver, &mdriver.pFull);
30 // [ version abrégée du fichier : les autres associations ne sont pas incluses ]
31    SysOSStart();
32    return 0;
33 }
```

Figure 2.7 : Fichier global (*main*) de l'exemple *PacketRouter*

Pour utiliser les modules préalablement codés, il faut d'abord inclure tous les fichiers de structure qui ont été programmés. Il faut également inclure le fichier *SysOS.h* qui contient les services associés au simulateur Syslib. Comme les modules sont des objets C++, on en crée des instances (lignes 9-11). Même phénomène pour les *SysChannel* qui sont des objets modèles, acceptant des types de données Syslib entre crochets. Ainsi, on pourra créer des canaux de données de type *SysInt* (ligne 20) et des canaux d'événements de type *SysEvent* (ligne 22). Comme les canaux de données possèdent une file d'attente, on

pourra en ajuster la taille (fixée à 1 par défaut) avec la méthode *setFifoDepth* (ligne 25). Une fois tous les canaux nécessaires créés, on les associe (*binding*) aux ports des modules. Il faut connecter le canal à deux ports provenant de modules différents. L'opération est possible grâce à la méthode *bind* en fournissant séquentiellement en paramètre le module de sortie, le port de sortie, le module d'entrée et le port d'entrée (lignes 28-29). Enfin, la simulation est contrôlée à l'aide des méthodes *SysOSStart* et *SysOSStop*.

2.5. Implantation de la bibliothèque

Nous verrons dans cette section les détails derrière l'implantation de Syslib, notamment ceux de la structure des modules, des ports et canaux, mais aussi ceux du simulateur Syslib qui gère tous ces modules. Nous mettrons l'emphase sur une section importante: la subtilité derrière les fonctions de rappel qui supportent les contextes d'exécution. Des ajouts ont été faits pour le support de la hiérarchie de modules et seront également présentés. Mais d'abord, nous allons examiner l'environnement ainsi que les fonctionnalités de base que tout programmeur devrait connaître pour mieux orienter ses spécifications.

2.5.1. Environnement

Syslib a été développée pour fonctionner sous tous les environnements Windows. La bibliothèque utilise des concepts classiques OO de C++, tels les classes, les modèles, l'héritage et le polymorphisme ainsi que la surcharge des opérateurs. Syslib utilise des structures de données provenant de la bibliothèque externe STL (*Standard Template Library*) [DeDe98] telles des listes, des files d'attentes et des ensembles. Les structures principalement utilisées seront décrites tout au long de cette section. STL offre une boîte à outils rapidement utilisable et valide pour tous les types de données. Enfin, Syslib a été développé sous l'environnement de développement *Visual C++* version 6 de Microsoft. Par ailleurs, une version Linux de la bibliothèque Syslib la rend encore plus attrayante pour un concepteur friand de cette plate-forme. Un port pour Linux a été ajouté et sert notamment à la création des *threads*.

2.5.2. Structure interne d'un module

La notion de module s'apparente à un processus élémentaire (*thread*) en logiciel ou à un processus (*process*) en matériel (VHDL). L'exécution de ces modules, nous l'avons mentionné, est non bloquante. Chaque module contient plusieurs unités d'exécution. Ces unités sont créées au moyen d'objets *SysBehaviour* (stockés dans un *map* STL). Les unités à exécuter sont soit **réactives**[3], soit **perpétuelles**. Chaque *SysBehaviour* est associé à une liste de signaux d'événements (qui constitue la liste de sensibilité) et contient une liste de **fonctions de rappel** à exécuter selon les événements notifiés. Les *SysBehaviours* perpétuels sont toujours exécutés en premier (en ordre de leur déclaration s'il en existe plus d'un). Ainsi, seules les unités d'exécution demandées sont lancées. Cette structure est décrite à la figure 2.8.

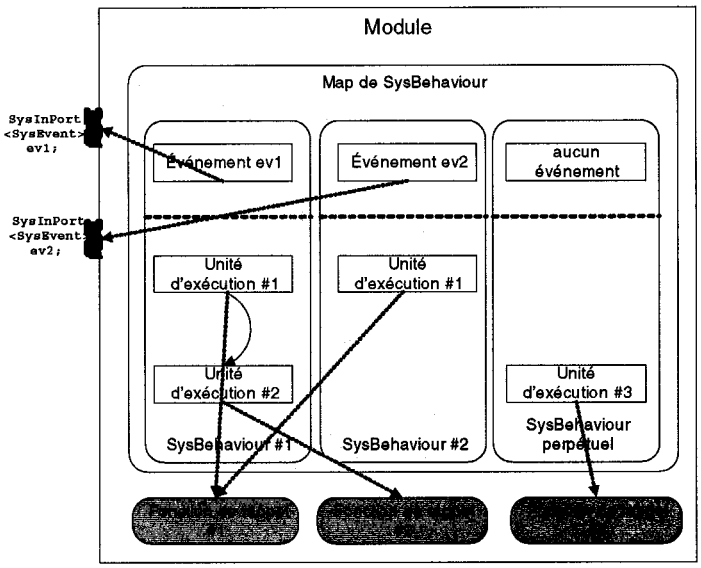


Figure 2.8 : Structure d'un module Syslib

Lorsqu'un événement est notifié, tous les modules attachés à cet événement sont ordonnancés et seules les unités concernées par cet événement seront exécutées. Si

[3] Qui sont régies par une liste de sensibilité et donc sensibles à des événements. On dira aussi en anglais *event-driven.*

certaines conditions ne sont pas rencontrées au cours de l'exécution, l'unité termine simplement (*return*) et donne la main au prochain *SysBehaviour* de ce module.

2.5.3. Structure interne des ports et canaux

D'abord, le port permet de communiquer des données à l'extérieur du module, évidemment vers d'autres modules. Le port peut être utilisé à l'interne comme une variable pour des opérations classiques (affectation, addition, soustraction, etc.) puisque les opérateurs des classes de ports ont été surchargés. Cela permet d'éviter la déclaration de variables tampons. Par conséquent, le port accède au canal pour la lecture ou l'écriture de données, ou encore pour la notification et la consommation d'événements. Cela apparaît comme une distinction claire entre l'algorithme et les communications extérieures. En cas d'erreur (aucune donnée disponible à lire dans le canal, écriture impossible car canal plein, notification impossible, etc.), le canal retourne un message au port que le programmeur peut intercepter pour réagir en conséquence.

Le canal représente le moyen de communiquer les informations entre les modules. On lui associe une file d'attente (FIFO) de taille configurable pour simuler l'existence d'une mémoire partagée entre deux modules et ainsi assurer l'intégrité des données lors d'un échange. On permet de créer un FIFO sur les canaux de données seulement. Il n'est donc pas permis d'accumuler les événements. Comme mentionné à la section 2.4, les données et les événements doivent être consommés. Lorsqu'on lit sur un canal, on consomme, i.e. on retire l'information du canal et on la transfère vers le port (la variable qui représente le port dans le module). La consommation s'applique également aux canaux de données de profondeur 1 (qui sont également implantés à l'aide des files d'attente). La consommation de l'événement permet au programmeur de clairement voir les entrées qui ont réveillé le module.

Il est permis de connecter un port de sortie à plusieurs entrées. Cela permet ainsi de propager une donnée ou un événement à plusieurs modules. Pour ce faire, il faut créer un canal pour chaque chemin de données ou d'événements. On peut établir la règle qu'il faut

un canal pour chaque port d'entrée. Il n'est cependant pas permis de connecter plus d'une sortie dans une entrée. Ceci est illustré à la figure 2.9.

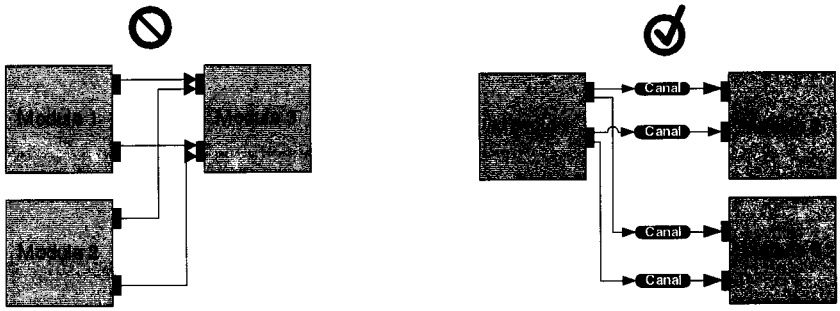


Figure 2.9 : Connexions non permises et permises avec Syslib

Ces connexions sont supportées par une structure de données rattachée à chaque port qu'on appelle *SysConnect*. Techniquement, cette structure relie un port au canal auquel il est connecté. De plus, le canal est responsable d'avertir le simulateur en cas de notifications d'événements. La structure des canaux et des ports est illustrée à la figure 2.10. La figure représente deux modules connectés dont les ports sont associés à des canaux de données et d'événements. Chaque type de canal est représenté. Le canal de données contient un FIFO pour le stockage des données. Également, le port contient une valeur tampon pour simuler la persistance.

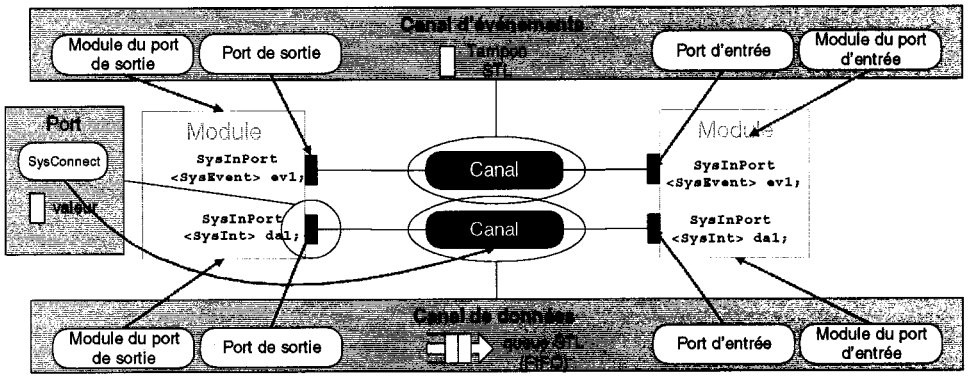


Figure 2.10 : Structure des ports et des canaux Syslib

2.5.4. Types de données abstraits

Les données et les événements de Syslib présentés au tableau 2.2 sont de types distincts et abstraits. La séparation des événements et données apporte la clarté dans le code. Les types de données proposés sont restreints à des types abstraits utiles pour un système, dont les booléens, les caractères et les entiers courts et longs, signés ou non. On peut également utiliser des structures contenant ces types. Au niveau fonctionnel, il n'est pas nécessaire d'avoir des vecteurs de bits à manipuler. Pour les événements, des objets particuliers ont été créés. L'événement est doté d'un numéro d'identification pour le simulateur, mais également d'une valeur binaire : 0 ou 1. Ainsi, on pourra activer un module à l'aide d'un 0 ou d'un 1, selon les besoins.

Tableau 2.2 : Types abstraits dans Syslib

Type	Description	Intervalle de valeurs[4]
`SysLong`	Donnée 32 bits signés	-2^{16} à $2^{16}-1$
`SysULong`	Donnée 32 bits non signés	0 à $2^{32}-1$
`SysInt`	Donnée 32 bits signés	-2^{16} à $2^{16}-1$
`SysUInt`	Donnée 32 bits non signés	0 à $2^{32}-1$
`SysShort`	Donnée 16 bits signés	-32768 à 32767
`SysUShort`	Donnée 16 bits non signés	0 à 65535
`SysChar`	Donnée 8 bits signés	-128 à 127
`SysUChar`	Donnée 8 bits signés	0 à 255
`SysBool`	Donnée 1 bit	*true* ou *false*
`SysEvent`	Événement	0 ou 1

2.5.5. Engin de simulation Syslib

L'engin de simulation de Syslib est une partie fondamentale de la bibliothèque. Son fonctionnement est inspiré de Cynlib. Sa structure se distingue par sa simplicité. Le développement de l'engin a donc été très rapide. L'engin a été présenté dans [FCBA02] et est repris à la figure 2.11. Cet engin de simulation utilise la bibliothèque STL et ses algorithmes de recherche pour le stockage des *threads*. Une exécution typique en 10 étapes de deux modules est présentée à la figure 2.11. Selon la légende présentée dans le

[4] Intervalle de valeur peut varier selon les systèmes.

bas de la figure, chaque *thread* possède trois états distincts : E (lorsqu'il est en exécution), S (lorsqu'il est suspendu et en attente d'être réveillé) et R (lorsqu'il est prêt à être ordonnancé pour son exécution). Les niveaux de gris présentent pour chaque étape les changements d'états d'un *thread* à survenir. L'exemple présente l'exécution de deux modules où un premier en exécution envoie des données à un second. Le premier module notifie par la suite le second et l'engin de simulation intercepte la notification pour mettre à jour la liste des modules à exécuter.

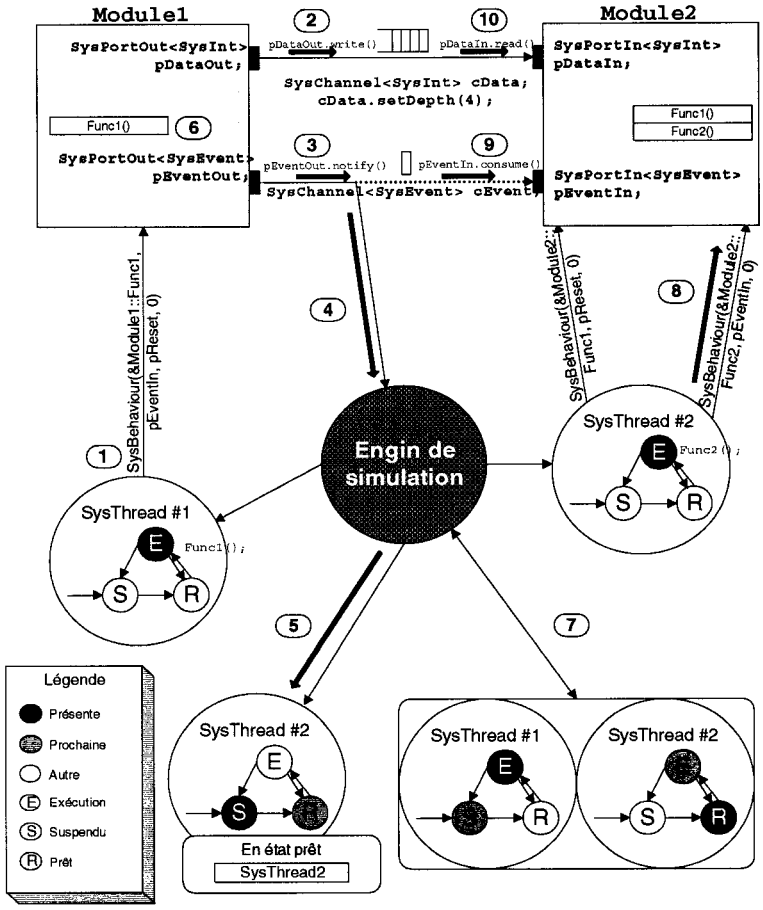


Figure 2.11 : Comportement de l'engin de simulation de Syslib

D'abord (1), le *thread SysThread1* lance l'unité d'exécution *Func1* du *Module1*. Ce module transfère (2) une donnée de son port vers le canal et notifie (3) un événement.

L'événement est capturé (4) par l'engin de simulation, qui fait balancer l'état de *SysThread2* de S à R (5). *Module1* termine son exécution (6). Le simulateur change (7) les états de *SysThread1* de E à S et de *SysThread2* de R à E. *SysThread2* exécute (8) alors les unités sensibles à l'événement notifié. L'événement est alors consommé (9) et les données sont lues (10).

Le simulateur fonctionne à l'aide d'un *thread* global pour tous les modules. Les modules sont ordonnancés par des mécanismes de manipulation de compteur programme (avec *setjmp/longjmp*).

2.5.6. Fonctions de rappels

Nous présentons ici quelques remarques au sujet des unités d'exécution (qu'on nomme aussi fonctions de rappel).

- Si un module (*thread*) contient des unités d'exécution perpétuelles, ces dernières seront exécutées avant les unités d'exécution réactives. À cause de la présence de ces unités perpétuelles, le module sera automatiquement réordonnancé à la fin de l'exécution de toutes les unités du module.
- La fonction de rappel d'un module qui est sensible à deux événements activés ne s'exécutera qu'une seule fois. Les deux événements devront être consommés.
- Si deux événements distincts sont reliés à deux fonctions de rappel distinctes d'un même module, le module ne sera ordonnancé qu'une fois. Par contre, les deux fonctions de rappel sont exécutées.
- Si d'autres événements destinés à un module sont activés lorsque celui-ci est déjà en attente d'exécution (à l'état R), le module n'est pas réordonnancé.
- Lors de la mise en marche du système, tous les modules possédant des comportements perpétuels sont ordonnancés.

2.5.7. Support de la hiérarchie de modules

La hiérarchie de modules est un concept essentiel pour un langage système. Elle permet d'abstraire la structure du design en cachant les détails d'implantation d'un module, mais

tout en conservant le même niveau d'implantation (dans notre cas, le niveau fonctionnel). Prenons l'exemple d'une unité arithmétique et logique (UAL) (figure 2.12a). Cette UAL est en fait composée d'un contrôleur et d'un chemin de données (figure 2.12b). Mais encore, on peut hiérarchiser ce chemin de données en une unité flottante et une unité entière (figure 2.12c).

Ce concept ajouté à Syslib permet de créer des modules dans des modules pour simplifier la conception de systèmes complexes et améliorer la clarté des systèmes conçus. L'ajout de cette fonctionnalité doit s'harmoniser avec les modules existants ainsi que les ports et les canaux utilisés pour concevoir les systèmes. La seule contrainte (pour des fins de simplicité d'implantation) est que les modules hiérarchiques ne contiennent que des modules et pas de code algorithmique. Sur la figure 2.12, le module hiérarchique du chemin de données ne contiendrait alors que le code servant à la création des modules des unités flottante et entière. Une implantation plus efficace aurait été de parcourir l'arbre des connexions avant la simulation afin de créer une liste des ports destinations ciblés par une écriture ou une notification. Nous définissons par ailleurs deux types de modules, les modules hiérarchiques (MH) qui ne cachent qu'une structure modulaire et les modules de premier niveau (MPN) qui implantent une fonctionnalité réelle. Il est à noter que les modules hiérarchiques doivent respecter les règles prédéfinies à propos des canaux et des connexions.

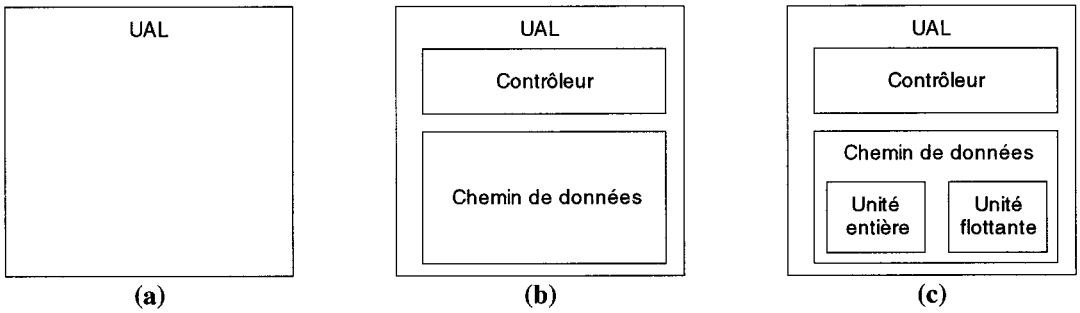


Figure 2.12 : Hiérarchie structurelle

La création de modules dans un MH a d'énormes implications quant à la connexion des modules internes au MH vers les modules externes au MH. Il faut associer les ports des modules internes à des ports du MH. Ainsi, les ports périphériques du MH pourront être reliés aux modules externes. Pour bien comprendre les implications au niveau des connexions des MH aux MPN, nous répertorions dans le tableau 2.3, les cas valides parmi les 16 cas théoriques de connexions possibles. Également, la figure 2.13 schématise ces connexions.

Tableau 2.3 : Énumération des cas de connexions valides

Numéro du cas	Direction du port d'origine	Structure du port d'origine	Direction du port cible	Structure du port cible
1	Sortie	MPN	Sortie	MH
2	Sortie	MPN	Entrée	MPN
3	Sortie	MH	Entrée	MH
4	Entrée	MH	Sortie	MH
5	Entrée	MH	Entrée	MH
6	Sortie	MH	Entrée	MPN
7	Sortie	MPN	Entrée	MH
8	Entrée	MH	Entrée	MPN
9	Sortie	MH	Sortie	MH

Pour supporter les modules hiérarchiques, on étendra la règle précédemment mentionnée à la figure 2.9 : seules les entrées d'un module de premier niveau (MPN) seront connectées à un canal. Les entrées d'un MH ne le seront pas. Cette restriction est également représentée à la figure 2.13.

Extension du support aux connexions

Pour supporter des connexions à des modules hiérarchiques, il faudra étendre le concept de *SysConnect* déjà présenté. Comme un port peut être attaché soit à un ou plusieurs ports hiérarchiques, soit à un ou plusieurs canaux ou un mélange des deux, il faut s'assurer de propager la valeur à tous les éléments branchés. Pour ce faire, chaque port contiendra une liste de *SysConnect* (*SysConnectList*). Un *SysConnect* devient soit un lien vers un canal,

soit un lien vers un autre port et le module auquel il appartient, tel que défini à la figure 2.14.

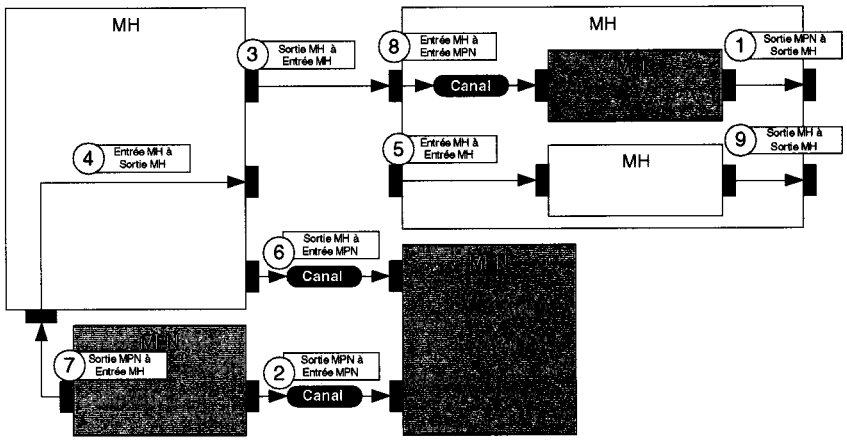


Figure 2.13 : Les 9 types de connexions valides avec Syslib

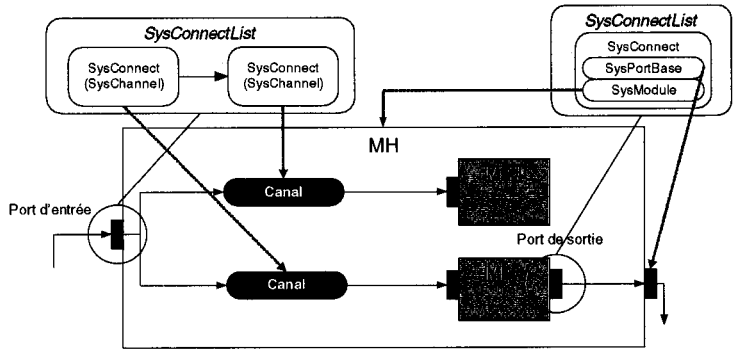


Figure 2.14 : Extension des *SysConnects*

2.5.8. Vue détaillée de l'implantation

Cette section fera part de concepts un peu plus avancés quant à l'implantation de Syslib. Elle répond à des questions fondamentales sur des éléments clefs derrière Syslib. Le schéma global UML simplifié de la figure 2.15 présente la structure des classes de Syslib.

Dans ce schéma, les boîtes représentent les classes définissant le projet. À l'intérieur de chaque boîte, on voit d'abord (dans la première case) le nom de la classe, suivi des attributs les plus importants (2^{e} case) et des méthodes les plus pertinentes (3^{e} case). Les lignes pleines se terminant par une flèche triangulaire indiquent la hiérarchie de classes (par exemple, *SysModule* est parent de *MyModule*). Les lignes pleines se terminant par un losange noir indiquent l'agrégation. On retrouve sur ces lignes le nombre d'agrégats permis pour chaque classe. Par exemple, pour 1 objet *MyModule*, on retrouve de 0 à * (infini) objets *SysInPort*. Également, pour 0 ou 1 objet *MyModule* on retrouve 0 à * objets *SysChannel*. Enfin, les lignes pointillées décrivent un lien de type *utilise*, principalement représenté par un pointeur. Par exemple, un objet *SysBehaviour* utilise un objet *SysEvent*. Les classes qui implantent Syslib sont décrites au tableau 2.4.

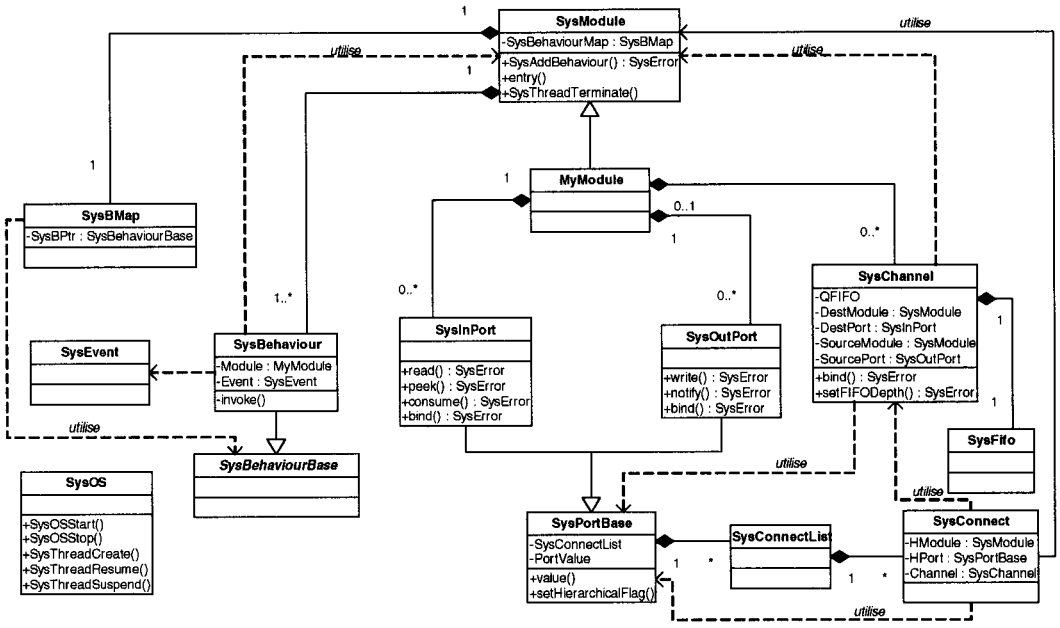


Figure 2.15 : Schéma des classes UML de Syslib

Tableau 2.4 : Classes du projet Syslib

Classe	Description
MyModule	Classe représentant le module du programmeur.
SysModule	Classe de base implantant le module dans Syslib. Tous les modules déclarés par un programmeur dérivent de ce module qui offre des services pour l'ajout d'unités d'exécution (avec *SysAddBehaviour*). Cette classe contient le SysBMap, ensemble des *SysBehaviours* du module. La méthode *entry* est le point d'entrée du *thread*. On peut terminer l'exécution d'une unité perpétuelle à l'aide de *SysThreadTerminate*.
SysBehaviour<M>	Classe modèle implantant un *SysBehaviour*. Le modèle M est un pointeur sur la fonction de rappel. La classe connaît l'événement et les fonctions de rappel à exécuter qu'elle appelle à l'aide de *invoke*.
SysBehaviourBase	Classe abstraite parent de la classe *SysBehaviour* permettant de manipuler les objets *SysBehaviour*.
SysChannel<T>	Classe modèle implantant le canal de communication. Le modèle T est le type de données du canal. Cette classe contient la file d'attente (*QFIFO*) par laquelle transite les données. Elle contient également des pointeurs sur les modules et ports source et destination qui lui ont été donnés par la méthode *bind*. On ajuste la taille des FIFO par la méthode *setFIFODepth*.
SysEvent	Classe implantant l'événement permettant de réveiller les modules.
SysPortBase<T>	Classe modèle parent aux classes de ports. Le modèle T est le type de données du port. Cette classe offre tous les services communs aux ports, dont la méthode *value* qui permet de retourner au module la valeur *PortValue* du port. La classe permet de configurer la hiérarchie du port avec *setHierarchicalFlag*. Enfin, chaque port contient une liste de *SysConnect* représentant les instances auxquelles il est rattaché.
SysPortIn<T>	Classe modèle représentant un port d'entrée. Le modèle T est le type de données du port. Elle offre les méthodes d'accès de lecture (*read* et *peek*) et de consommation (*consume*). La méthode *bind* permet d'associer un port d'entrée à un port hiérarchique.
SysPortOut<T>	Classe modèle représentant un port de sortie. Le modèle T est le type de données du port. Elle offre les méthodes d'accès en écriture (*write*) et en notification (*notify*). La méthode *bind* permet d'associer un port d'entrée à un port hiérarchique.
SysOS	Classe représentant l'engin de simulation de Syslib. Les méthodes publiques *SysOSStart* et *SysOSStop* permettent de démarrer et d'arrêter une simulation. Les méthodes suivantes servent à l'interne pour ordonnancer les modules.
SysBMap, SysConnect SysConnectList, SysFifo SysError	Classes représentant les structures de données internes utilisées par la bibliothèque Syslib.

Communications hiérarchiques

Les lectures de données et consommations d'événements se font toujours au premier niveau, puisque les canaux sont toujours rattachés aux MPN. En ce qui a trait aux écritures et aux notifications d'événements, le principe est différent : on parcourt une liste de *SysConnect* pour propager les sorties à toutes les entités connectées. Puisque ces

communications doivent se terminer dans un canal pour que la valeur puisse être stockée, on propage la valeur de façon récursive tant qu'on ne rencontre pas un canal qui termine le chemin de données. Lorsqu'on rencontre un port hiérarchique, on rappelle la fonction de propagation appropriée (*write* ou *notify*). La structure hiérarchique ne sert qu'à l'utilisateur pour simplifier la vue du système. Au niveau du simulateur, on élimine donc cette hiérarchie et on travaille sur un même niveau sans hiérarchie.

Exécution des fonctions de rappel

La complexité de l'exécution des fonctions de rappel réside dans le fait qu'elles sont des méthodes de classes, i.e. qu'elles sont représentées par deux pointeurs de 32 bits (un objet et une méthode). Lors de la création du *thread* du module, un contexte provenant du système d'exploitation[5] demande un pointeur (32 bits) de la fonction de *thread* à appeler, ce qui cause un problème puisque les fonctions de rappel en Syslib sont sur 64 bits. On opte alors pour la création d'une fonction d'entrée statique dans chaque module. Une fonction statique est définie comme en C sur un seul pointeur de 32 bits qui peut être passé au contexte du système d'exploitation. Une fois entré, on redirige l'exécution vers une fonction membre connue pour l'exécution du *thread*.

Les objets *SysBehaviour* sont des modèles C++ dont le type est un pointeur sur la fonction de rappel (méthode de classe) nommée par l'utilisateur. Ces objets sont impossibles à manipuler à l'interne car leur type est inconnu. On les manipule donc à l'aide d'une classe parent virtuelle (*SysBehaviourBase*). Il s'agit là d'un patron de design proposé dans [Alex01]. La création d'une table de fonctions virtuelles (*vTable*) a été suggérée, mais cela impliquait beaucoup d'effort de programmation. De plus, les modules possèdent généralement peu de fonction de rappel (une ou deux), ce qui met en doute la nécessité d'une telle table. Des explications complètes sur les pointeurs de fonctions (ou *Functors*) sont disponibles dans [Händ02].

---

[5] La création d'un contexte diffère selon le système d'exploitation (Windows et Linux) et on doit tenir compte pour assurer le fonctionnement de la bibliothèque sur deux systèmes d'exploitation.

2.6. Hiérarchisation de l'exemple

L'exemple du *PacketRouter* de la figure 2.4 est très simple. Si l'on regarde plus en détail le module *Driver* qui ne fait que vider la file d'attente, ce travail semble, dans les faits, peu réaliste. Pour que les paquets soient utilisables, le pilote devra non seulement lire les paquets, mais aussi analyser leur intégrité, tenter de les corriger dans les cas d'erreurs puis les réordonner en supposant leur arrivée dans le désordre. Pour réduire la complexité du système, on représenterait le *Driver* par un module hiérarchique, comme à la figure 2.16.

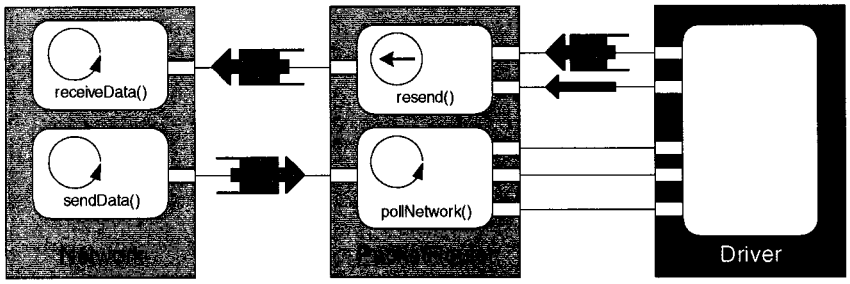


Figure 2.16 : Le module hiérarchique *Driver*

Le nouveau module hiérarchique *Driver* cache d'autres modules qui définissent son nouveau comportement dans le système. La figure 2.17 montre le contenu du module hiérarchique *Driver*. Nous allons démontrer ici l'impact de la création de MH en exposant une courte section de code décrivant le contenu du MH.

Le fichier du MH (*Driver*) contiendra les instances de tous les modules internes et des canaux les reliant. Les différences découlent principalement des associations (*binding*) des ports du MH aux ports internes associés. Les noms des principaux éléments en cause de cet exemple sont montrés à la figure 2.17. Le code décrivant une partie de leur implantation (notamment les connexions) est présenté à la figure 2.18. Il faudra consulter [Fili02] pour plus de détails concernant les modules hiérarchiques ainsi que des exemples complets couvrant toutes les possibilités.

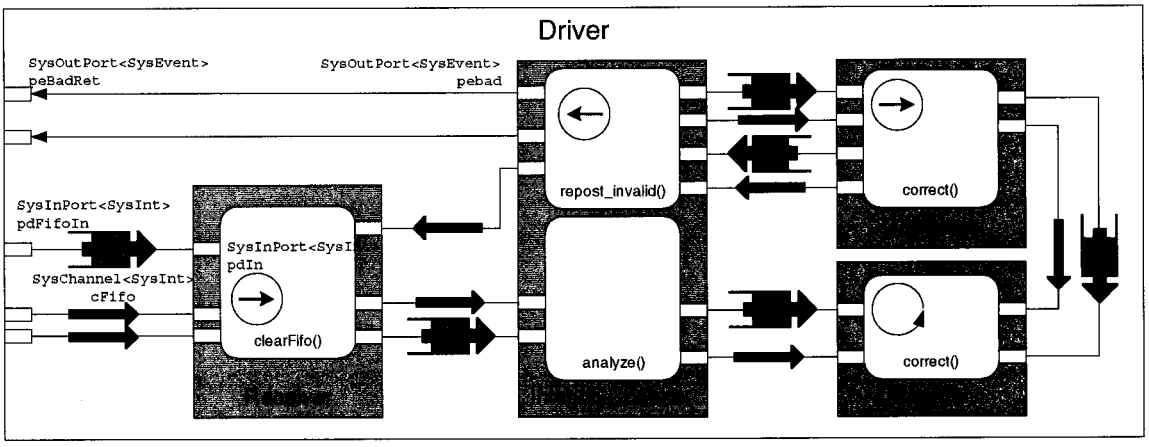


Figure 2.17 : Contenu du module hiérarchique *Driver*

```
1   Receiver receiver;
2   IntegrityChecker intcheck;
3
4   SysChannel<SysInt> cFifo;
5
6   peBadRet.setHierarchicalFlag(true);
7   pdFifoIn.setHierarchicalFlag(true);
8
9   // ÉTAPE 1 : associer les canaux aux ports
10  cFifo.bind(&driver, &driver.pdFifoIn, &receiver, &receiver.pdIn);
11
12  // ÉTAPE 2 association aux ports hiérarchiques de SORTIE
13  intcheck.pebad.bind(this, &peBadRet);
14
15  // ÉTAPE 3 associations aux ports hiérarchiques d'ENTRÉE
```

Figure 2.18 : Exemple de code pour l'implantation d'un module hiérarchique

Le simulateur doit différencier les ports hiérarchiques des ports de premier niveau. Les lignes 6 et 7 permettent d'affecter un drapeau aux ports hiérarchiques. Une fois les objets créés, trois étapes claires s'imposent. D'abord, il faut associer les ports aux canaux internes (ligne 10), comme c'est le cas pour le canal *cFifo*. La deuxième étape consiste en l'association des ports hiérarchiques de sortie. Une méthode *bind* implantée à même les ports peut être utilisée. On associe les ports en appelant la méthode du port de niveau hiérarchique le moins élevé. Une dernière étape consiste en l'association des ports hiérarchiques d'entrées (inexistant dans cet exemple). Ce cas surviendra au niveau global, lors de l'instance du module *Driver* et de sa connexion avec le module *PacketRouter*.

L'ajout de la hiérarchie de modules complique quelque peu la création de la structure du système, c'est pourquoi la vigilance est de mise et le simple fait de suivre ces quelques étapes permet d'éviter les oublis. Un élément intéressant est que la création de la structure d'un système se fait automatiquement avec l'outil Picasso, ce qui élimine les erreurs probables de cette fastidieuse tâche qu'est la connexion de tous ces modules. Picasso est présenté en Annexe E.

CHAPITRE 3
Analyse de l'implantation

Nous verrons dans ce chapitre les possibilités et les particularités qu'offrent la bibliothèque Syslib pour la conception des systèmes. Un autre point important consiste au raffinement des spécifications vers les niveaux d'abstraction inférieurs. Ce cheminement sera de plus en plus utilisé au cours des années à venir et il importe de démontrer la façon dont il s'applique. Ce chapitre présente d'abord une approche théorique du raffinement. Nous verrons par la suite le travail à accomplir pour des raffinements basés sur les quatre niveaux d'abstraction proposés par SystemC. Également, nous verrons un exemple concret de raffinement vers du matériel en Cynlib.

3.1. Apport de Syslib au design de systèmes

Le concepteur qui choisit une bibliothèque parmi d'autres sait que les services offerts par cette bibliothèque influenceront le design en cours. Malgré le fait que plusieurs bibliothèques sont en C++, les modèles de calcul et les concepts qui régissent ces bibliothèques ne sont pas tous les mêmes. Ainsi, Syslib possède certains aspects particuliers comme il a été mentionné dans la section précédente qui serviront à modeler un design d'une façon bien particulière : la façon Syslib.

3.1.1. Design non bloquant

D'abord, l'orientation non bloquante de Syslib affecte énormément la façon de spécifier le système. Syslib n'offre pas de blocages sur les FIFO, ni d'attente active (il n'offre aucune notion de temps). En général, le comportement naturel du logiciel consiste en l'exécution concurrente de *threads* reliés à l'aide de files d'attente (FIFO) bloquantes. Ainsi, une écriture (par un *thread*) dans un FIFO plein bloque ce *thread*, de sorte que ce dernier passe de l'état d'exécution à l'état d'attente. Il demeure dans ce mode tant que l'espace voulu (dans le FIFO) n'est pas disponible. Il en va de même pour une lecture bloquante dans un FIFO vide. On pourrait éviter ces blocages en écriture avec des files

d'attente de taille infinie (comme le veut le modèle de calcul *Kahn Process Network* ou KPN), mais cela suppose des ressources mémoire infinies qui n'existent pas. Par conséquent, le fait que Syslib ne bloque en aucun cas implique que tout module s'exécutant doit terminer son exécution et recommencer du début à chaque ordonnancement. De ce fait, on peut qualifier le modèle de calcul de Syslib comme étant un KPN simplifié. Ce phénomène influence la structure finale du code et lui donne des aspects orientés contrôle, tels une machine à état.

Le design d'un module de décompression de Huffman (pour un décodeur JPEG) communiquant avec une table de valeurs (*look-up table*) est un bon exemple. Son algorithme général est présenté à la figure 3.1. Des explications plus détaillées sur le décodeur JPEG seront présentées au chapitre 4.

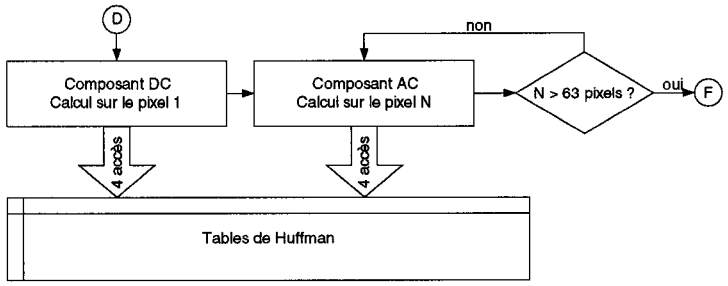


Figure 3.1 : Algorithme général de la décompression de Huffman

Dans cet exemple, le traitement débute avec le calcul d'un composant DC pour un bloc de pixels. Pour ce calcul, 4 constantes doivent être extraites d'une table de valeurs ne possédant qu'un port d'accès; l'opération totalise donc 4 accès. On passe ensuite au calcul de 63 composants AC de ce même bloc de pixels où, à nouveau, 4 accès en table par pixel sont requis. La figure 3.2 présente les versions bloquante et non bloquante pour ce même algorithme.

Version bloquante

```
; Composant DC
Demande Constante HUFFSIZE
wait()
Demande Constante MAXCODE
wait()
Demande Constante MINCODE
wait()
Demande Constante HUFFVAL
wait()

; Composant AC
for (N = 1 TO 63)
{
   Demande Constante HUFFSIZE
   wait()
   Demande Constante MAXCODE
   wait()
   Demande Constante MINCODE
   wait()
   Demande Constante HUFFVAL
   wait()
}
```

Version non bloquante

```
; Composant DC
if (component_state == DC)
   if (state == HUFFSIZE)
      Demande Constante Huffsize
   if (state == MAXCODE)
      Demande Constante MaxCode
   if (state == MINCODE)
      Demande Constante MinCode
   if (state == HUFFVAL)
      Demande Constante Huffval

; Composant AC
if (component_state == AC)
   if (pixel < 63)
      if (state == HUFFSIZE)
         Demande Constante Huffsize
      if (state == MAXCODE)
         Demande Constante MaxCode
      if (state == MINCODE)
         Demande Constante MinCode
      if (state == HUFFVAL)
         Demande Constante Huffval
      pixel = pixel + 1
   else
      fin
```

Figure 3.2 : Exemples d'algorithmes bloquant et non bloquant

Les deux versions fonctionnent parfaitement, mais une version est plus rapide à programmer et certainement moins dense. Dans la version bloquante, on insère des *waits* pour créer une attente active laissant le temps de charger la constante d'un autre module. À l'éveil, l'exécution reprend du point où l'attente a débuté. Dans le cas de la version non bloquante, les *waits* n'existent pas et l'exécution redémarre du début après chaque constante demandée. Il faut donc sauvegarder l'état du système pour reprendre l'exécution là où l'on avait laissée. La boucle de l'étape *Composant AC* de la version bloquante ne sera exécutée qu'une seule fois, même si le module lui-même sera ordonnancé plus de 250 fois. Dans la version bloquante, les variables locales conservent leurs valeurs lors des changements de contexte, ce qui n'est pas le cas pour la version non bloquante qui demande l'utilisation de variables globales ou de membres de classe pour conserver l'état du système. La version non bloquante est plutôt orientée vers une implantation matérielle, contrairement à la version bloquante qui décrit nettement du logiciel. Or, il existe des programmes de synthèse comportementale qui transforment du

code bloquant vers le matériel, mais le programmeur doit suivre plusieurs règles strictes et styles afin que cela soit possible. Ces synthétiseurs comportementaux évolueront au cours des prochaines années.

3.1.2. Ordonnancement

L'ordonnancement des modules Syslib s'effectue d'une façon bien précise qu'il est utile de connaître avant le codage des modules. Au démarrage, les modules possédant des unités d'exécution perpétuelles sont toutes ordonnancées. Si aucun module n'en contient, alors il faudra « forcer » la notification d'un événement, mais ceci n'est pas recommandé. Un démarrage correct serait l'utilisation d'un module *Reset* possédant une unité perpétuelle qui lance un événement. Comme les modules possédant des unités perpétuelles (ou modules perpétuels) se réordonnancent automatiquement après leur exécution, il est possible de les « tuer » à l'aide d'un appel à *SysThreadTerminate*. Cependant, *Reset* pourrait être conservé pour relancer le système en cas d'erreurs d'exécution. Un exemple d'ordonnancement pour un système simple est présenté à la figure 3.3.

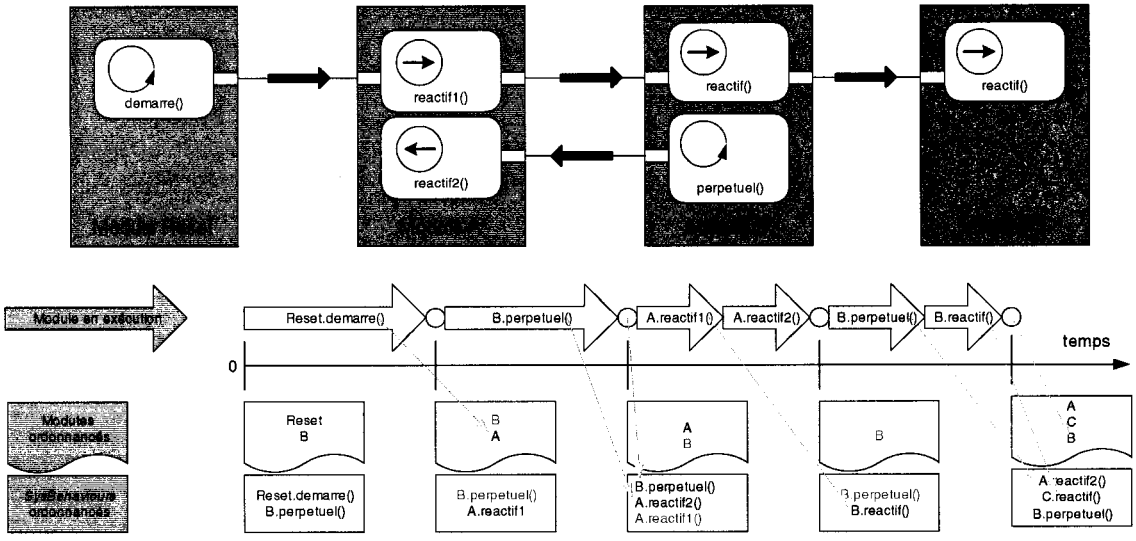


Figure 3.3 : Ordonnancement des modules et *SysBehaviours*

Dans cet exemple, on voit sur la ligne du temps les modules en exécution (par les flèches contours) qui influencent l'ordonnancement. Ainsi *Reset.demarre()* notifie un événement qui ordonnance le module A et son *SysBehaviour reactif1*. *Reset* ne reviendra plus par la suite, puisqu'on le désactivera (par un appel à *SysThreadTerminate*). Nous voyons que le module perpétuel B se réordonnance à la fin de l'exécution de tous ses comportements (représentée par le cercle vide sur la ligne de temps).

Par ailleurs, Syslib propose une façon simple de contrôler l'ordonnancement de ses modules. Cette manière consiste à créer un module perpétuel qui s'occupe lui-même de réveiller tous les autres modules (en étant connecté à eux par un événement). Il peut ainsi récupérer certaines informations et ordonnancer en conséquence les modules nécessitant une exécution. Cette technique peut s'avérer utile dans le cas de petits systèmes ou de parties de systèmes plus imposants, mais est déconseillée pour un système global.

3.2. Raffinement d'une spécification

Le raffinement consiste en la transformation progressive d'une spécification vers des niveaux d'implantation plus détaillés. Chaque étape de raffinement dévoile un peu plus la structure finale et les opérations du SoC en développement. Il n'est pas dit qu'on passera d'une spécification de haut niveau sans notion de temps directement à une implantation matérielle RTL ou encore logicielle. Chaque étape de raffinement apporte son lot d'information à analyser, pour permettre de bien partitionner un système.

Le raffinement touche plusieurs notions des composants d'un système. La première consiste en la transformation d'une spécification séquentielle pour y insérer la concurrence des modules; on parle alors de raffinement atomique [HoLF02]. Le raffinement consiste ensuite à ajuster l'algorithme lui-même qui compose l'exécution des modules. Pour un raffinement vers le matériel on voudra qu'une spécification soit synthétisable, c'est-à-dire qu'on puisse transformer sa description en une liste d'interconnexions représentative du circuit logique de la spécification. Le raffinement aborde également la délicate question des communications. En effet, une communication

de haut niveau impliquant un lien direct (ou avec FIFO logiciel) ne se compare en rien avec la demande d'accès à un bus pour amorcer une transaction. Les protocoles utilisés durant les communications se raffinent également, alors qu'un accès mémoire fonctionnel se décompose en une multitude de signaux, synchrones ou non, à piloter. Un avantage important de la séparation des communications de l'algorithme principal permet de gagner beaucoup de temps sur le raffinement des communications, notamment en se servant d'interfaces pour implanter les protocoles d'échange. Ainsi, un concepteur n'a qu'à interchanger ces interfaces pour raffiner les communications et évaluer plusieurs protocoles, sans modifier le code de l'algorithme. Cette méthode, on le sait, permet une meilleure intégration de blocs IP, mais permet aussi d'introduire dans un système des IP à différents niveaux d'abstraction. Finalement, les types de données sont également à raffiner, où l'on passe par exemple d'un type logiciel entier tout usage à un vecteur de bit précis en matériel.

Lorsqu'il est question de la direction du raffinement, on s'inspire souvent de modèles liés à l'architecture sous-jacente sur laquelle l'application sera implantée. Ainsi, le raffinement des spécifications tend vers des éléments spécifiques d'exécution, tels les coprocesseurs mathématiques dédiés, les microcontrôleurs, etc. Les communications doivent reproduire le comportement architectural des mémoires partagées, registres à décalages, lignes d'interruptions, bus avec arbitrage, etc. C'est entre autre ce que propose [GoGB97] pour SpecC où 4 types d'architectures cibles prédéfinies dirigent le raffinement.

3.3. Raffinement à niveaux multiples

Comme SystemC est devenu le standard *de facto* de l'industrie avec ses 4 niveaux d'abstractions : UTF, TF, BCA et PCA, il apparaît pertinent d'approfondir le raffinement selon ces niveaux d'abstraction. La figure 3.4 montre le chemin à suivre lors du raffinement des spécifications. C'est en partance d'une spécification UTF que nous raffinons vers les niveaux inférieurs existants. Dans [FiBA02c], nous avons établi que des spécifications de système en Syslib^{FL} étaient de niveau UTF. C'est pourquoi nous

survolerons le raffinement spécifique avec Syslib^{FL}, i.e. les traits pointillés sur la figure, qui montrent le raffinement vers du logiciel pur et du matériel en Cynlib.

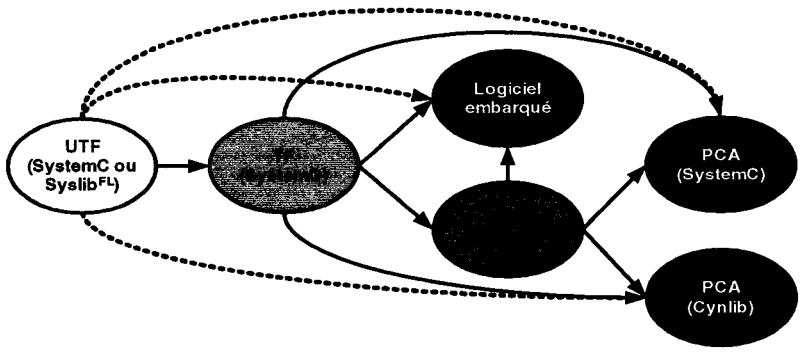


Figure 3.4 : Chemins possibles de raffinement

3.3.1. Vers le niveau temporisé (TF)

Cette première étape de raffinement conduit les spécifications sans notion de temps à un niveau dit temporisé qui permet d'évaluer le temps d'exécution des modules et des communications. Mais d'abord, un premier travail de raffinement devrait permettre de réviser la spécification de façon à pouvoir mieux évaluer le poids des opérations. À titre d'exemple, on propose dans [HoLF02] de transformer le calcul d'une racine carrée (calculée à l'aide d'une fonction de bibliothèque C++) en une procédure utilisant la méthode de Newton. On peut ainsi évaluer plus efficacement le délai de traitement.

Raffinement de Syslib^{FL}

Comme la bibliothèque Syslib^{BL} n'est pas disponible, nous devons considérer le raffinement des spécifications programmées en Syslib^{FL} vers la bibliothèque SystemC pour y ajouter des détails de niveau TF. [DGOS01] présente une méthode pour permettre l'interopérabilité des spécifications, i.e. la simulation conjointe de modules provenant de bibliothèques différentes. Cette technique pourrait être appliquée à Syslib.

Exemple de partitionnement

Pour évaluer la distribution des délais à insérer au niveau de l'exécution et des communications, il importe d'avoir une référence fiable. Cette référence est l'implantation de la spécification sur une architecture ciblée. Selon un partitionnement fictif ou suggéré, on peut ainsi faire une évaluation grossière des délais de chaque module en fonction de la fréquence d'horloge du processeur embarqué ou du matériel hôte. Il importe également d'établir un bref estimé pour les communications en évaluant le nombre de transactions entre les modules et leur type: des transactions sur bus, broche à broche ou par l'intermédiaire d'une mémoire partagée ou de registres dédiés. Un exemple de système – un décodeur JPEG, d'abord programmé avec SystemC UTF puis raffiné vers SystemC TF – démontre ce procédé. Le raffinement dans ce cas ne consiste qu'en l'introduction d'attentes temporisées par la fonction *wait(délai)* de SystemC. Pour évaluer les délais à introduire, un partitionnement fictif a pris lieu sur la plate-forme TarP [Bert03] selon la figure 3.5.

Les modules du décodeur JPEG seront présentés au chapitre 4. Sur l'architecture, les modules partitionnés en logiciel (*M_PROC* et *M_IHUFFMAN*) communiquent par un bus à une mémoire locale ou par un pont (*bridge*) AMBA à une mémoire lointaine partagée. Les communications des modules matériel implantés, soit sur un DSP (*M_IDCT*) ou en logique dédiée (*M_IQUANT*), passent par un bus AMBA arbitré et relié à une mémoire partagée. Les communications entre le logiciel et le matériel passent donc par cette mémoire. Certains modules (*M_IHUFFMANTABLE*, *M_IQUANTTABLE*) qui représentent des tables de valeurs précalculées sont situés dans leurs mémoires respectives. Pour réduire le trafic sur le bus commun, on propose une optimisation en ajoutant un registre de 64 pixels pour les communications entre le module *M_IQUANT* implanté sur logique dédiée et le module *M_IDCT* sur le DSP.

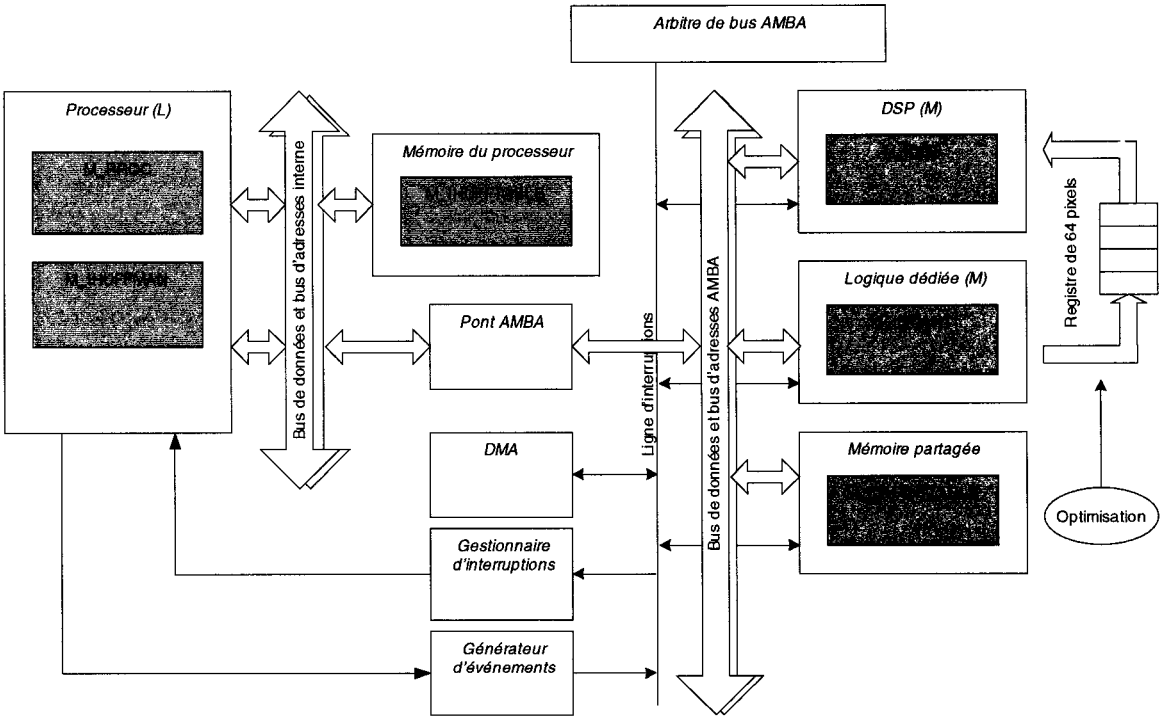


Figure 3.5 : Partitionnement d'un décodeur JPEG sur la plate-forme TarP

Une fois les modules partitionnés (de façon fictive bien sûr), il s'agit d'identifier les communications existantes et d'affecter un temps raisonnable à chacune d'elles. Le tableau 3.1 présente ces évaluations dans le cas de la plate-forme TarP. Ces cas estiment les accès au bus AMBA incluant une attente moyenne de quelques cycles par pixel, de même que l'accès au pont AMBA qui cause un certain délai. Les opérations arithmétiques sont temporellement estimées selon l'entité sur laquelle elles sont implantées.

Les résultats obtenus par cette opération d'exploration architecturale sont bien entendu préliminaires. Les communications sur bus se feront fort possiblement en rafale (*burst mode*) puisque le décodeur JPEG échange des données rassemblées en blocs de 64 pixels, ce qui influencera sans doute l'évaluation du temps d'exécution suggéré par opération. Les résultats peuvent également être utilisés pour raffiner l'approche pipelinée présentée en Annexe C. La simulation finale permet d'obtenir un estimé de la performance des

modules en exécution sur une architecture. On pourra ainsi recueillir de l'information plus tôt pour mieux contrer les problèmes de performance et engendrer de meilleures idées de partitionnement.

Tableau 3.1 : Répertoire des opérations avec leur temps d'exécution estimé

Module	Nombre d'opérations (par pixel traité)	Temps d'exécution suggéré par opération	Total
M_PROC	4 écritures en mémoire locale 1 lecture en mémoire lointaine Traitement	1 cycle 6 cycles (incluant l'attente bus) 10 cycles	20 cycles
M_IHUFFMAN	4 lectures en mémoire lointaine 8 additions 8 décalages 4 lectures en table 1 écriture en mémoire lointaine	3 cycles 1 cycle 1 cycle 3 cycles 6 cycles (incluant l'attente bus)	46 cycles
M_IDCT	1 lecture en mémoire ou *1 lecture en registre spécial* 16 multiplications en parallèle avec 15 additions 1 écriture en mémoire	4 cycles (incluant l'attente bus) *1 cycle* 4 cycles 4 cycles (incluant l'attente bus)	72 ou *68* cycles
M_IQUANT	1 lecture en mémoire 1 multiplication 1 écriture en mémoire *1 écriture en registre spécial*	4 cycles (incluant l'attente bus) 4 cycles 4 cycles (incluant l'attente bus) *1 cycle*	12 ou *11* cycles
M_IHUFFTABLE *M_IQUANTTABLE*	*Modules passifs implantés en mémoire*	0	0

3.3.2. Vers le logiciel

Le raffinement des spécifications partitionnées en logiciel est pratiquement nul. Il consiste à ouvrir la voie à des services logiciels : sémaphores, variables partagées, priorités de *threads*, mais également il doit s'attacher à une table d'adresses pour identifier les modules matériels destinataires. Ces services sont fournis par un RTOS embarqué sélectionné. Idéalement, nous devrions utiliser à ce niveau les mêmes interfaces de programmation (API) qu'aux niveaux supérieurs. Pour le module logiciel, la communication vers les autres modules (matériel ou logiciel) doit être transparente ou ne requérant que des modifications mineures. Aussi, toutes les communications devraient être supportées par le RTOS. Par exemple, le processeur (sur lequel le logiciel s'exécute)

peut adresser un générateur d'impulsions pour créer des événements vers un module matériel. Mais aussi, les modules logiciels en exécution peuvent être interrompus par un gestionnaire d'interruptions, lui aussi encapsulé par le RTOS.

3.3.3. Vers le niveau transactionnel (BCA)

Ce niveau intermédiaire permet de mieux évaluer les impacts de l'architecture choisie sur les spécifications. Ce raffinement touche principalement les communications, puisqu'on y modélise les comportements du bus (et de son arbitre). Les communications deviennent des transactions sur un bus et des cycles de communications sont introduits. Ces cycles peuvent être basés ou non sur une horloge, selon la précision attendues des résultats. Dans [GLMS02], on présente un exemple simple de bus décrivant les opérations à faire pour créer des transactions avec SystemC.

3.3.4. Vers le niveau matériel (PCA)

Un raffinement vers le niveau matériel (PCA) pour décrire une spécification synthétisable est un travail majeur. Il est probable qu'on veuille également transformer la spécification au niveau RTL, ce qui est fort complexe même si le langage utilisé demeure le C++. Pour une transformation vers le niveau PCA, on sépare généralement les modules en deux éléments distincts, un chemin de données (*datapath*) et un contrôleur de style machine à états, ce qui constitue un changement algorithmique important. Les modules deviennent sensibles aux changements des fronts d'horloge et à des signaux de synchronisation. L'insertion de vecteurs de bits fait également parti du raffinement. Si l'objectif visé est celui de la synthèse, on réduit le plus possible les chemins de données en modules plus simples : registres, multiplexeurs, éléments de logiques simples, etc.

3.3.5. Exemple de raffinement matériel avec Cynlib

Le niveau Cynlib est équivalent au niveau matériel (PCA) SystemC décrit ci-dessus. La méthodologie proposée au chapitre 2 guide la spécification Syslib partitionnée en matériel vers la bibliothèque Cynlib. Des modifications au niveau des constructions du programme sont requises, notamment les boucles et les appels aux fonctions des

bibliothèques C++. D'autres signaux de contrôle dédiés doivent être ajoutés pour régler les problèmes de synchronisation typiquement rencontrés. Bref, tous ces changements sont directement reliés aux modules matériel et nécessitent une large part de travail. Des modèles matériel intermédiaires non synthétisables ainsi que des bancs de tests seront aussi créés pour valider ces raffinements. Heureusement, la synthèse comportementale [Elli99] permet de demeurer à un niveau d'abstraction plus élevé que le RTL ce qui avantage de plus en plus les solutions par raffinement progressif.

La traduction des spécifications de Syslib à Cynlib est facilitée par la similitude de la syntaxe et de la structure derrières ces deux bibliothèques. La figure 3.6 met en opposition les structures Syslib et Cynlib des fichiers d'en-tête pour le module *PacketRouter* du chapitre 2.

```
1  #include "Syslib.h"
2
3  class PacketRouter : public SysModule {
4  public :
5    PacketRouter();
6
7    SysInPort<SysInt>     pFromNetwork;
8    SysOutPort<SysInt>    pToNetwork;
9    SysOutPort<SysInt>    pFifo;
10   SysInPort<SysInt>     pPacket;
11   SysInPort<SysEvent>   pOnResend;
12   SysOutPort<SysEvent> pFull;
13   SysOutPort<SysEvent> pHalf;
14
15 private :
16   void pollNetwork() ;
17   void resend() ;
18 };
19
20
21
22
23
24
25
26
```

```
#include "cynlib.h"

class PacketRouter : public CynModule {
public:
 PacketRouter(char* name, In<1> sClk,
    In<1> sReset, Out<32> sNetOut,
    Out<1> sPutNet, In<32> sNetIn,
    Out<1> sGet, In<1> sResend,
    In<32> sSWIn, Out<32> sSWOut,
    Out<1> sPutSW);

 In<1>   pClk;
 In<1>   pReset;
 Out<32> pNetOut;
 Out<1>  pGet;
 In<32>  pNetIn;
 Out<1>  pPutNet;
 In<1>   pResend;
 Out<1>  pPutSW;
 In<32>  pSWIn;
 Out<32> pSWOut;

 void assign();
 void reset();
 void resend();
};
```

Figure 3.6 : Structures Syslib et Cynlib du module *PacketRouter*

À gauche, la structure du module en Syslib et à droite, la structure avec Cynlib. On constate à prime abord du côté Cynlib que le constructeur reçoit en paramètre des *sockets* pour se connecter à des ports externes au module (lignes 5-10). On constate que les ports

s'identifient comme des *In<>* et des *Out<>*, avec entre crochets la largeur du port en bits. On remarque également l'ajout de ports pour l'horloge et pour un signal de réinitialisation (lignes 12-13) suivis de trois prototypes de fonctions de rappel.

Au niveau de l'implantation, on créera des processus répondant aux fronts d'horloge: on transformera les *AddSysBehaviour()* de Syslib en des *execute_on()* Cynlib. On utilisera aussi des assignations différées. Les ports faisant circuler des structures de données peuvent utiliser les constructions Cynlib *InST<>* et *OutST<>*. La traduction de l'un à l'autre peut être automatisée, une fois tous les cas possibles identifiés, à l'aide d'outils d'analyse syntaxique (*parser*). L'Annexe D présente brièvement l'implantation d'un tel outil.

Au niveau interconnexions Syslib, les modules sont reliés entre eux à l'aide de canaux de données et d'événements. On peut facilement remplacer le canal d'événement par un signal de contrôle Cynlib de 1 bit. À l'opposé, le canal de données et son FIFO sont indubitablement plus difficiles à remplacer. Les échanges entre modules logiciel passent par la mémoire locale sans souci pour le programmeur, mais les échanges de données matériel-matériel, matériel-logiciel ou logiciel-matériel doivent transiter par une unité de stockage quelconque. L'introduction de FIFO Cynlib facilite le raffinement; c'est pourquoi la création d'un module nommé *CynFifo* pour relier les instances matérielles et logicielles est préconisée. De ce fait, si l'on partitionne le *PacketRouter* en matériel, on introduira à l'exemple trois *CynFifos*, tel que démontré à la figure 3.7.

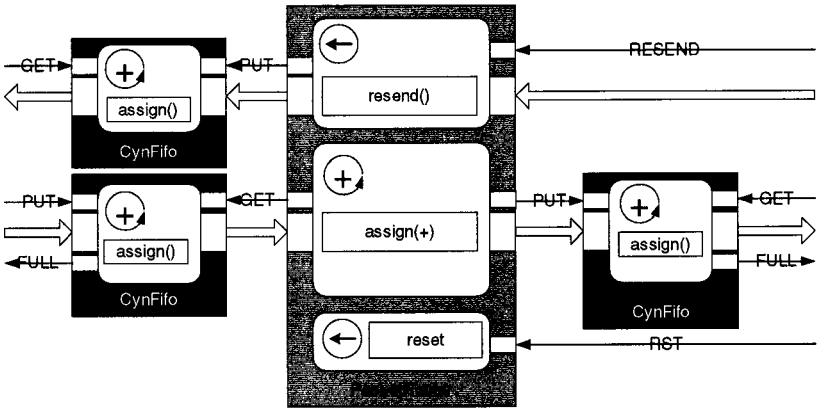


Figure 3.7 : Utilisation des *CynFifos* lors du raffinement matériel

Le schéma de la figure utilise un gabarit similaire à celui de Syslib où plusieurs des unités d'exécution (processus) sont réactives aux fronts d'horloge positifs.

CHAPITRE 4
Analyse des résultats

Pour bien mesurer ses capacités et comprendre son fonctionnement, il est fondamental d'utiliser Syslib pour implanter quelques systèmes ou exemples de design typiques afin d'en recueillir des résultats quantitatifs et qualitatifs. Cette tâche permettra également de bien saisir les limites de la bibliothèque Syslib. Cinq exemples de design de petite et moyenne envergure ont été développés et sont présentés dans ce chapitre: il s'agit d'un additionneur simple, d'un contrôleur de mémoire, d'un *BlockMatcher*, du décodeur JPEG déjà entraperçu au chapitre 3, et finalement, nous ferons un retour sur le routeur de paquet déjà présenté au chapitre 2. Tous ces exemples ont d'abord été programmés avec l'aide de la bibliothèque Syslib et le code de tous ces exemples peut être téléchargé de [Fili02b].

Comme il a été préalablement exposé, SystemC prend une place importante sur le marché et la version 2.0.1 supporte bien la conception système. Il apparaît intéressant et motivant de comparer les capacités de Syslib à celles de SystemC. Pour ce faire, tous les exemples de design énumérés ci-dessus ont également été programmés avec SystemC en utilisant le niveau d'abstraction UTF. Le code peut similairement être récupéré de [Fili02b]. De ce fait, une comparaison des résultats obtenus avec Syslib et SystemC est légitime et nécessaire pour mieux situer le travail accompli avec Syslib.

Les résultats seront présentés de façon progressive tout au cours de ce chapitre et des explications sur SystemC seront également apportées en chemin lorsqu'il apparaîtra pertinent de le faire. Pour chaque exemple, les différences majeures entre SystemC et Syslib et les principales observations seront énumérées. Les résultats quantitatifs et qualitatifs applicables à tous les exemples suivront.

4.1. Procédures

4.1.1. Mesures de performance

Pour les résultats quantitatifs, on a implanté dans la bibliothèque Syslib un mode de débogage qui calcule les temps de communications séparément : écritures, lectures, notifications, etc. Le temps est calculé sous Windows avec le compteur de performance haute résolution [Micr97] qui est plus précis comparativement au compteur C ANSI/ISO de *time.h* utilisé dans la version Linux. Les mesures de performances présentées dans ce mémoire ont donc été effectuées sous Windows. Tous les temps affichés représenteront le nombre d'incréments du compteur de performance Windows que nous appellerons *unités de temps* ou *UT*. Les expériences de mesures acheminées tiennent compte de l'erreur de la mesure elle-même. Dans les cas où le temps nécessaire pour mesurer la performance d'une opération, de lecture par exemple, est avoisinant de la résolution du compteur de performance, nous avons mesuré le temps pour un grand nombre d'exécutions, puis nous avons divisé le temps total par le nombre d'itérations.

4.1.2. Gabarits

Les exemples de ce chapitre qui sont représentés graphiquement utiliseront principalement le gabarit de design Syslib présenté au chapitre 2. Dans le cas du décodeur JPEG, un gabarit proposé pour SystemC et présenté à la figure 4.1 sera utilisé.

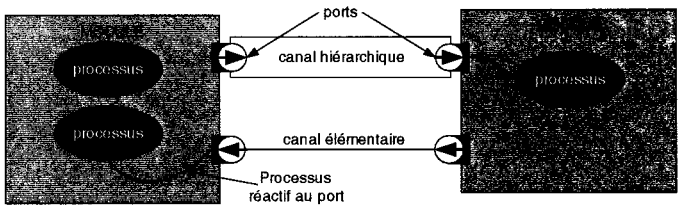


Figure 4.1 : Gabarit SystemC

Il est tiré d'une proposition de [Schw02] (et simplifié pour les besoins de ce mémoire). Le schéma résultant est moins volumineux, ce qui procure une meilleure saisie visuelle de ce système. Il sera utilisé aussi dans le cas de l'additionneur simple. Nous n'utiliserons

pour ces exemples SystemC que les canaux élémentaires, c'est-à-dire les canaux déjà implantés dans SystemC [GLMS02].

4.2. Exemple #1 : l'additionneur simple

Toute bibliothèque utilise des exemples simples pour expliquer ses fonctionnalités et l'additionneur est souvent un exemple de choix. Il s'agit d'un additionneur fonctionnel qui génère, à partir d'un module, deux entiers à additionner et les envoie à un second module qui les additionne et retourne le résultat.

4.2.1. Implantation

L'implantation de ce système simple est décrite à la figure 4.2a. La version Syslib comporte deux modules de 5 ports chacun. Ces modules sont connectés par 5 canaux. Le module de gauche, *Generator*, lance les deux valeurs aléatoires, A et B, jumelées à un événement qui servira à ordonnancer le module *Adder*. Ce dernier fait l'addition et retourne le résultat C jumelé d'un événement au premier module. Le module *Generator* possède deux unités d'exécution: d'abord *generate* qui est perpétuelle et qui génère les nombres à additionner, puis *display* qui est réactif à l'événement qui passe sur *done*. Le nom et le type des canaux sont indiqués à même la figure. Le même algorithme, cette fois programmé avec SystemC, est défini à la figure 4.2b qui démontre l'utilisation du gabarit SystemC.

4.2.2. Différences avec SystemC

Nous remarquons d'abord que l'additionneur en Syslib comporte plus de ports que celui en SystemC. Les modules SystemC peuvent être ordonnancés d'après un changement sur n'importe quel port d'entrée, ce qui n'est pas le cas avec Syslib qui se doit d'être réveillé par un événement (le type abstrait *SysEvent*).

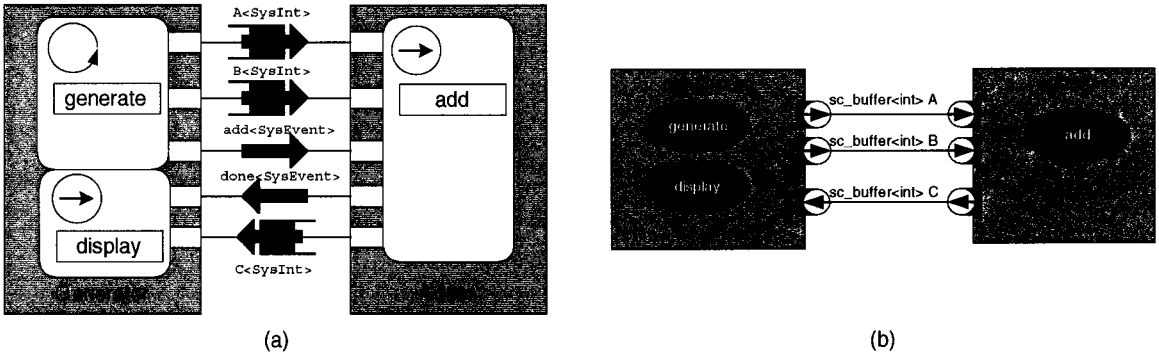


Figure 4.2 : Exemple de l'additionneur simple avec Syslib et SystemC

4.3. Exemple #2 : le routeur de paquets

L'exemple du routeur de paquets est présenté au chapitre 2 et sert d'introduction à la bibliothèque Syslib. Veuillez consulter ce chapitre pour revoir les explications algorithmiques du routeur.

4.3.1. Différences avec SystemC

Le premier problème qui apparaît est que SystemC ne peut reproduire le comportement des modules perpétuels de Syslib. Un comportement similaire peut être obtenu en SystemC, lorsqu'on introduit des processus (*SC_THREAD, SC_CTHREAD*) qui s'ordonnancent après un intervalle de temps donné ou selon une horloge. L'insertion de notions de temps change le niveau d'abstraction du système en faisant théoriquement passer le modèle du niveau UTF (sans temps) au niveau TF (avec temps). Cette transformation pourrait causer des problèmes lors du raffinement des phases subséquentes, principalement pour évaluer correctement les délais préalablement introduits.

Il a été dit en présentant cet exemple au chapitre 2 que le routeur de paquets possédait un comportement perpétuel *pollNetwork* qui simulait un comportement d'interrogation pour une unité d'exécution particulière. Pour les mêmes raisons, on peut affirmer que l'interrogation n'est pas un comportement possible avec SystemC de niveau UTF.

4.4. Exemple #3 : le contrôleur mémoire

Un contrôleur de mémoire qui relie ordinairement un processeur à une mémoire est orienté pour le contrôle. Ce genre d'unité est principalement implantée en matériel sur les SoC. Il apparaît intéressant cependant de pouvoir simuler son comportement à haut niveau pour plus de performance de simulation lors des phases préliminaires de design.

4.4.1. Implantation

Le contrôleur implanté et illustré à la figure 4.3 possède des fonctionnalités typiques : il simule des requêtes mémoire de lecture ou d'écriture de tailles différentes destinées vers des unités mémoires différentes. Ce contrôleur est connecté à différents modules qui peuvent être considérés comme des bancs de tests. Le module *Processeur* génère et envoie les requêtes aux trois différents modules centraux qui représentent le contrôleur lui-même. Ces modules auraient pu être intégrés dans un module hiérarchique mais ils ont été exposés pour les fins de l'exemple. Ces modules sont *Wordsize* qui définit la taille de l'accès (octet, demi-mot ou mot), *ChipSelect* qui active la destination visée (la mémoire, le générateur d'événements et le gestionnaire d'interruptions) et enfin *StateMachine* qui gère le type d'accès (lecture ou écriture) ainsi que le chemin de données. Le module *Mem* implante la mémoire pour répondre aux requêtes et un module *Sink* a été implanté pour recevoir les événements destinés aux autres modules non importants ici (le gestionnaire d'interruptions et le générateur d'événements).

Tous les modules de cet exemple possèdent une unité d'exécution sauf *Processeur* qui en possède deux dont une perpétuelle nommée *begin* qui démarre la simulation. Le module *StateMachine* en possède également deux, dont *trans*, qui est perpétuelle et qui fait transiter les données de requêtes sur les canaux à chaque ordonnancement.

À titre d'exemple, les modules du contrôleur transforment une requête de lecture, en un signal de sélection de puce (*chip select* ou *CS*) et un signal de validation de sortie mémoire (*output enable* ou *OE*) de même qu'un signal pour la taille de l'accès (*byte*

enable ou BE) pour le module *Mem.* Des canaux existent pour faire circuler les données du processeur à la mémoire (ou vice-versa). Une amélioration possible pour cet exemple serait d'implanter un mode rafale (*burst mode*) et d'allonger la file d'attente des canaux de données.

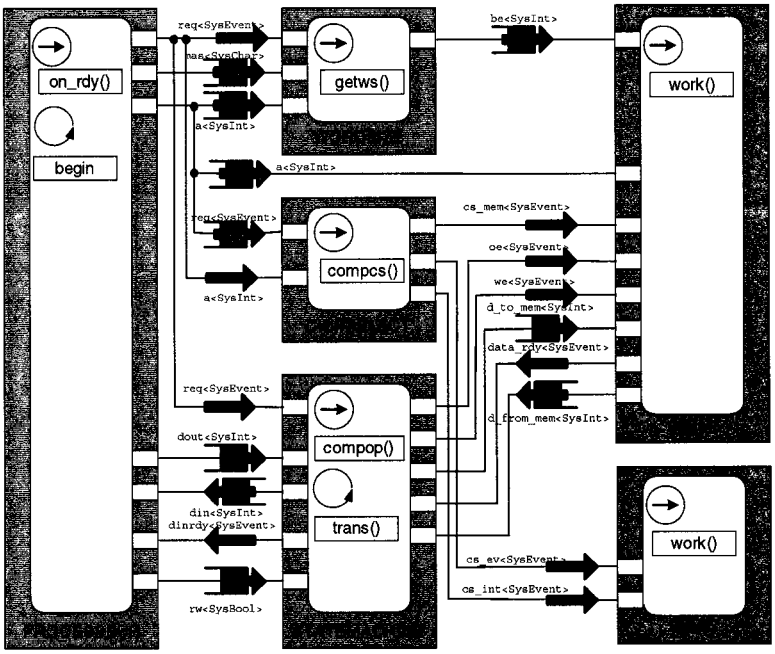


Figure 4.3 : Exemple du contrôleur mémoire

4.4.2. Différences avec SystemC

Après la transformation en SystemC et l'exécution de la spécification, plusieurs différences ressortent au niveau du comportement des unités d'exécution. Premièrement, il est impossible en SystemC de connaître le port qui a réveillé les unités d'exécution d'un module, lorsque cette unité est sensible à plusieurs ports d'entrée. Il est possible de connaître l'émetteur d'un *sc_event* mais selon notre expérience, ces événements ne sont utilisables qu'à l'intérieur d'un module, pour communiquer d'un processus à l'autre. Une façon simple de procéder avec SystemC est alors d'associer un seul signal réactif par processus. Par conséquent, les fonctionnalités globales d'un module risquent d'être dispersées sur plusieurs fonctions, ce qui peut mener à un manque de compréhension du programme. Dans Syslib, le concept d'événements distincts et leurs consommations

obligées clarifient la lecture du code. De plus, on peut discerner les événements qui ont réveillé les unités d'exécution, ce qui est essentiel à ce niveau d'abstraction pour analyser des algorithmes rapidement.

Avec Syslib, toutes les valeurs sont envoyées par FIFO, forçant le module à les consommer, i.e. les retirer du canal et les conserver dans le port du module ou dans une variable registre temporaire. SystemC offre un comportement similaire par le canal élémentaire *sc_fifo*. D'autres canaux élémentaires existants et souvent utilisés, *sc_signal* et *sc_buffer*, ne sont pas orientés pour la consommation des données. La valeur propagée sur ces canaux demeure la même tant que le module émetteur ne la modifie pas. Conséquemment, tous les modules doivent mettre à jour leurs sorties lors de leur exécution. Ceci ne semble pas si terrible, mais peu typique d'un paradigme de modélisation de haut niveau. Pis encore, la réécriture de toutes les sorties génère un réordonnancement de tous les modules connectés. Des modules ordonnancés inutilement s'exécutent, vérifient une condition fausse et terminent, ce qui allonge considérablement le temps de simulation.

Enfin, le concept de modules perpétuels en Syslib permet au programmeur de bien organiser son code en rassemblant des parties communes des unités d'exécution au même endroit et à un moment d'exécution connu. La même chose pourrait être implanté avec SystemC, à l'aide d'appels de fonctions, mais la construction de code pour faire les appels au bon moment semble complexe.

4.5. Exemple #4 : le *BlockMatcher*

Le *BlockMatcher* fait partie intégrante des applications d'encodage vidéo MPEG et sert à trouver des correspondances sur la position de blocs de pixels sur une séquence vidéo. Le *BlockMatcher* est inclus dans un algorithme complexe nommé *motion estimation*. Plus d'informations sont disponibles dans [Watk98].

Prenons un exemple simple : un soleil se lève à l'horizon sur la mer. Un bloc de pixels pourrait encadrer le soleil qui se déplace vers le haut avec le jour qui avance (le temps).

Comme le contenu du bloc de pixels qui contient le soleil ne change pas, on peut stocker le bloc, puis son déplacement plutôt que de stocker la nouvelle image. Cette approche diminue la taille du fichier final. Ceci est un exemple simplifié, un vrai *BlockMatcher* s'appliquerait évidemment pour toutes les images d'une séquence vidéo pour limiter la taille d'un fichier vidéo sur disque (ou à télécharger). Le *BlockMatcher* a été substantiellement simplifié pour cet exemple. Cette version est inspirée de [BeFi01]. Ce *BlockMatcher* comprend une unité dont la fonction est d'envoyer une série d'images (16 pixels par 16 pixels) et un patron (8 pixels par 8 pixels) à détecter dans chacune des images. Le *BlockMatcher* fait l'évaluation du déplacement du patron dans les images reçues. À partir des déplacements recueillis au fil des images, il est possible d'établir le vecteur déplacement du bloc dans la séquence d'images. Ce vecteur compresse 4 déplacements sur un entier.

4.5.1. Implantation

Le *BlockMatcher* a été implanté de différentes façons. D'abord, il supporte la hiérarchie de modules Syslib, tel que démontré par le module *Encodeur* de la figure 4.4. Les détails de ce module sont révélés à la figure 4.5.

L'exemple est implanté en utilisant la technique du module perpétuel qui ordonnance les autres modules au moment voulu. Dans ce cas précis, l'ordre d'exécution est connue au départ et il s'agit d'un ordonnancement statique. Cette technique pourrait également être utilisée pour un ordonnancement dynamique. Pour ce faire, le module *Perpetuel* doit recueillir des informations en cours d'exécution pour décider de l'ordonnancement à générer en fonction de l'exécution.

La simulation commence avec le module *Générateur* qui génère les images et le patron, les envoie au *Détecteur* qui cherche la position du patron dans l'image. À la figure 4.5, on calcule les déplacements du patron dans l'image (*Compresseur*) qu'on envoie au *Vectorisateur* qui les stocke dans un seul entier. Lorsque le vecteur est plein, il est envoyé au module *Afficheur* qui l'affiche à l'écran.

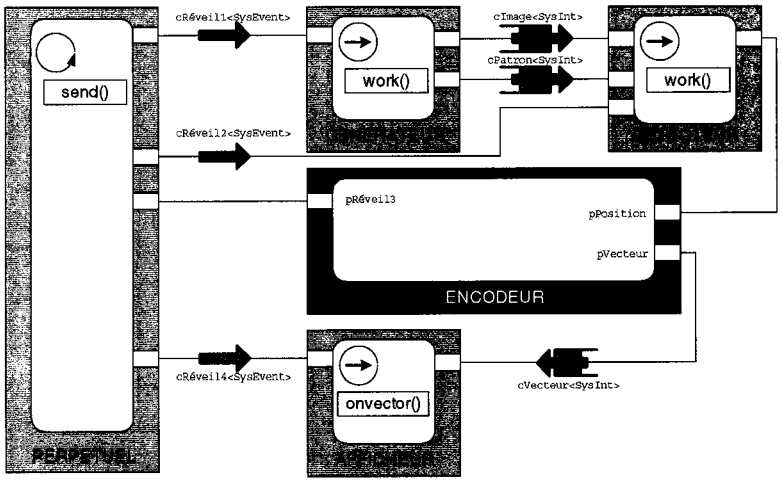


Figure 4.4 : Exemple du *BlockMatcher*

Au niveau des communications, deux versions différentes ont été implantées. D'abord, une version comportant un canal de données avec FIFO, puis une seconde version où l'on a utilisé, au lieu du FIFO, un tableau de ports qui représente un bus de données. Par exemple, au lieu d'un canal de 256 entiers de profondeur pour le transfert de l'image de *Générateur* à *Détecteur*, on aura un tableau de 256 ports de profondeur 1 chacun.

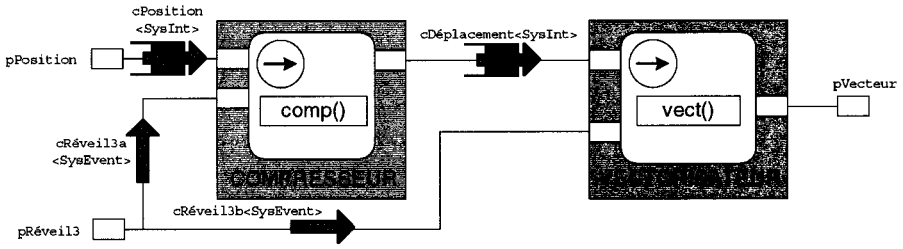


Figure 4.5 : Détails du module *Encodeur* du *BlockMatcher*

4.5.2. Différences avec SystemC

Cette technique pour contrôler l'ordonnancement avec SystemC n'est pas vraiment possible au même sens qu'avec Syslib, à moins d'avertir spécifiquement et au bon

moment (par un événement) le module qui fait l'ordonnancement. Il pourrait être possible d'utiliser la surveillance globale de SystemC qui observe de façon permanente si une condition particulière est vérifiée et qui ordonnance un certain module lorsque c'est le cas, mais cela semble être assez complexe à réaliser.

4.6. Exemple #5 : Le décodeur JPEG

Le décodage d'une image JPEG est une opération longue et complexe. Le but de ce mémoire n'est pas d'en décrire toutes les opérations, mais il apparaît essentiel d'en décrire la structure de base pour bien comprendre les éléments en cause. Pour de plus amples informations concernant les aspects théoriques du décodage ou encore pour obtenir plus de détails sur le décodage ou l'encodage JPEG, consultez les documents [Wall91] et [PeMi93].

L'algorithme du décodage JPEG s'effectue en 3 étapes claires telles que démontrées à la figure 4.6. On dirige une image JPEG compressée vers une première étape de décompression entropique par la méthode de Huffman. La décompression se fait à l'aide de tables de symboles stockées dans le fichier source. Une fois décompressés, les pixels sont regroupés par blocs de 8x8 et passent par une quantification inverse qui sert à redonner aux échantillons un nombre de bits minimum pour leur représentation. Pour cela, des valeurs sont lues à partir des tables de quantification inverse. Les blocs de pixels sortant sont envoyés vers la DCT inverse qui transforme la représentation des pixels du domaine fréquentiel au domaine temporel. Il en sort des blocs de pixels décompressés, généralement dans un format de données brutes YUV.

Le format YUV est un espace colorimétrique au même titre que le RGB et est particulièrement utilisé en télévision. L'information de base est Y, la luminance du pixel à laquelle on ajoute de l'information de chrominance U et V. Pour plus d'informations sur les espaces colorimétriques, voir [Jack02].

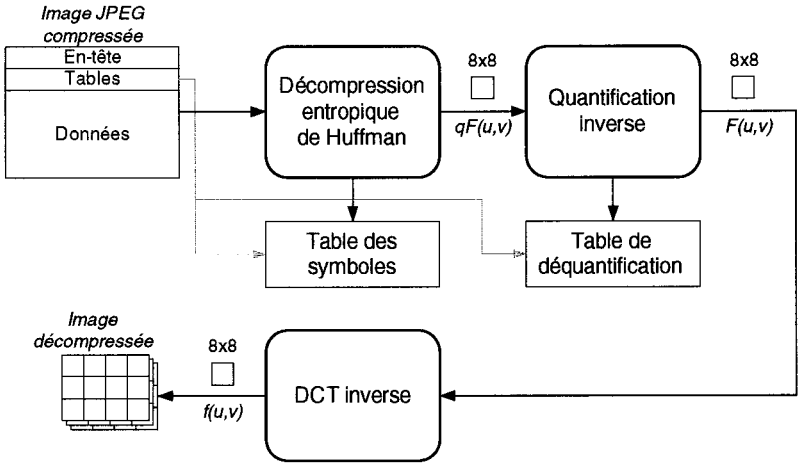


Figure 4.6 : Algorithme de décodage JPEG

Toute mention de bloc dans cet exemple fait référence à un assemblage de 8 pixels par 8 pixels. On utilisera pour cette implantation du décodeur JPEG le format YUV 4:2:0 affiché à la figure 4.7, où à 4 blocs de luminance sont jumelés 2 blocs de chrominance (Cb et Cr). L'image finale est interpolée à partir de l'information existante.

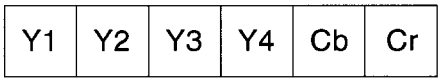


Figure 4.7 : Format d'image YUV 4:2:0

Les blocs décodés sont transformés vers l'espace colorimétrique RGB pour être sauvegardés dans un fichier en *bitmap* (.BMP).

4.6.1. Première version d'implantation

Une première implantation en Syslib et SystemC consiste en une structure de 6 modules, très ressemblante à l'algorithme de base. La figure 4.8 décrit cette structure en utilisant le gabarit de SystemC présenté à la figure 4.1[6]. On y trouve d'abord un module *M_PROC* représentant un processeur qui contrôle les opérations de décompression. À partir du processus *begin*, on lit le fichier JPEG à décompresser. On en extrait les tables de décompression de Huffman et de quantification inverse qu'on envoie respectivement aux deux modules *M_IHUFFMANTABLE* et *M_IQUANTTABLE*. Le module *M_PROC* envoie ensuite les valeurs de l'image à décompresser au module *M_IHUFFMAN* qui les récupère dans le processus *sequential_huffman* et les décompresse tout en accédant à la table des symboles préalablement stockée dans *M_IHUFFMANTABLE*. Le module *M_IHUFFMAN* crée des blocs de 64 pixels qu'il envoie au module *M_IQUANT*. Ces pixels sont lus dans *inverse_quantize*, puis la quantification inverse a lieu à l'aide des tables stockées dans *M_IQUANTTABLE*. Une fois le bloc traité, il est envoyé au module *M_IDCT* qui les récupère dans le processus *perform_IDCT*. Ces blocs décompressés et décodés sont ensuite renvoyés au module *M_PROC* qui les intercepte par son processus *collect_decoded_data* pour les transformer en RGB et les enregistrer dans un fichier sur le disque. Chacun des modules de tables possède deux processus, un qui répond aux communications avec le module *M_PROC* (pour le stockage des tables en début de simulation) et un autre pour répondre aux requêtes d'accès aux tables.

Cette version implantée possède une particularité qui démontre bien les possibilités du niveau fonctionnel sans notion de temps. Pour chaque bloc de pixels traité, les tables pour la décompression et la quantification inverse sont transférées au complet dans les modules principaux, et ce même si on n'a besoin que d'une seule valeur dans ces tables.

---

[6] Le même système décrit avec le gabarit de Syslib est plus difficile à lire car il contient beaucoup plus de ports. Pour fins de clarté, nous présentons seulement le décodeur JPEG avec le gabarit SystemC.

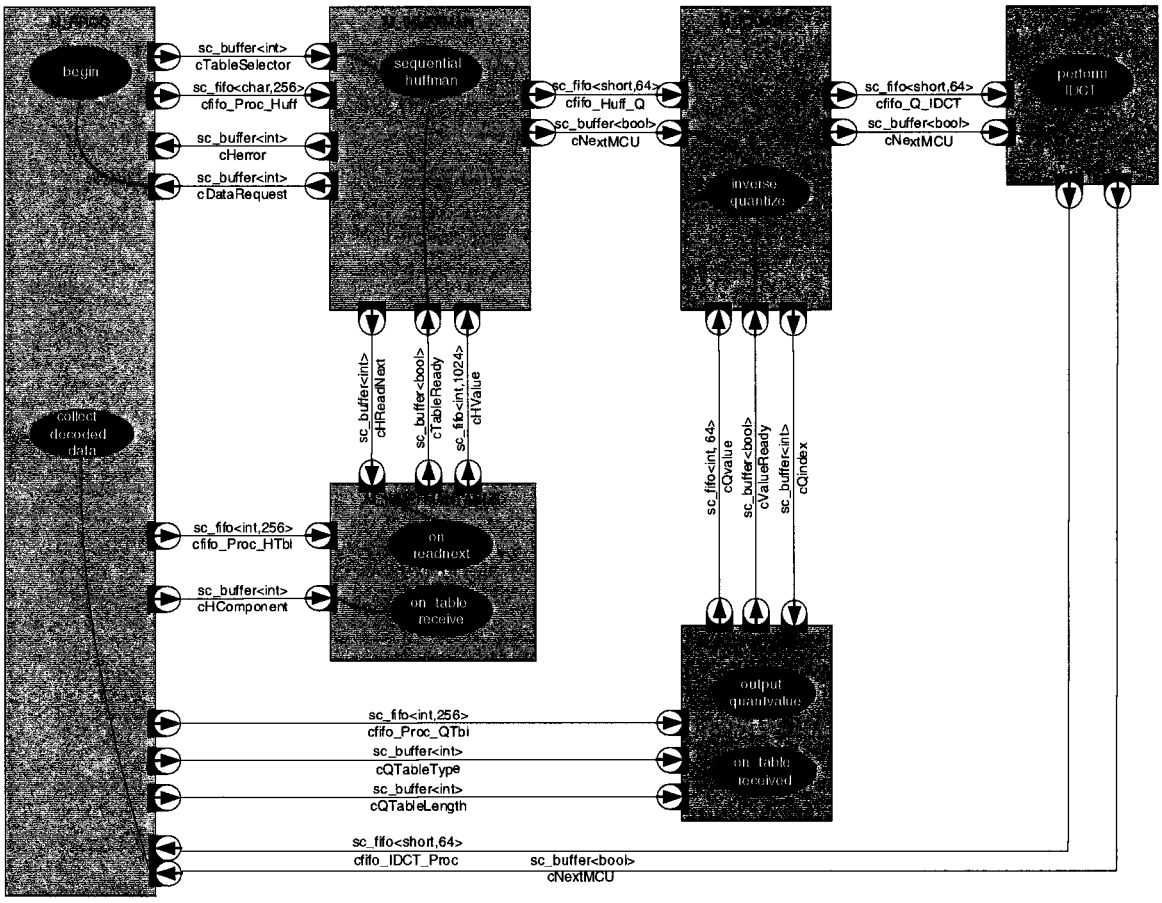


Figure 4.8 : Structure SystemC du décodeur JPEG

Cette façon de faire provient d'un premier raffinement qui a rendu le décodeur JPEG modulaire. Ce décodeur existait au départ en C pur et avait accès à toutes les tables de façon globale. Il apparaît aberrant de transférer les tables en entier pour chaque bloc, mais cela est tout à fait acceptable puisqu'il s'agit d'une exécution sans notion de temps et que tous ces transferts prennent un temps 0 à l'exécution. Par ailleurs, cette version implante un décodeur JPEG qu'on peut pipeliner puisque tous les étages d'exécution sont parfaitement balancés. Ce pipeline est étudié à l'Annexe C.

4.6.2. Deuxième version d'implantation

Il va sans dire qu'il est impensable d'implanter l'algorithme précédant tel qu'il est. Nous allons par conséquent modifier la première version pour limiter les communications au strict minimum et ainsi obtenir un système plus réaliste : seules les données nécessaires

seront transférées lors des accès aux tables. Cette modification aura pour effet de ralentir l'exécution (ce qui est provoqué par une augmentation des changements de contexte). Les modules traitant la décompression de Huffman seront ordonnancés beaucoup plus souvent, ce qui brise le pipelinage observé. Cette nouvelle implantation implique des ajouts de signaux, de drapeaux et de ports.

4.6.3. Différences avec SystemC

Plusieurs distinctions ressortent de ces implantations, sans doute parce que le nombre de lignes associé à cet exemple est grand. L'implantation Syslib du décodeur est beaucoup plus complexe, principalement parce que l'algorithme de décodage JPEG est orienté vers le traitement des données : des milliers de blocs circulent à travers les modules durant la décompression. Le module *M_IHUFFMAN* provoque beaucoup d'attente active à cause de nombreux accès aux tables de symboles, ce que Syslib ne peut supporter. Comme spécifié au chapitre 3, la structure des unités d'exécution doit être modifiée dans un style de machine à état ou séparée en de multiples modules et unités d'exécution pour pouvoir supporter tous ces accès.

La version SystemC utilise des *SC_THREAD* qui ont besoin de boucles infinies. Dans la version Syslib, on doit retirer ces boucles. Il faut également créer des variables membres pour remplacer les variables locales qui sont perdues lorsqu'une unité termine son exécution. Cela peut sembler simple en apparence, mais il en est d'un tout autre ordre lorsqu'on regarde par exemple, le noyau du décodeur JPEG qu'on retrouve dans le module *M_PROC* de la version SystemC et ses appels bloquants, versus la version non bloquante de Syslib.

La figure 4.9 montre le noyau de l'implantation avec SystemC. Cette version a été simplifiée pour ne conserver que la structure principale qui nous importe ici. L'image à parcourir est séparée en un certain nombre de blocs de base (MCU) parcourus selon les index i et j. Dans chacun de ces blocs de base peut se trouver des composants de luminance ou de chrominance parcourus selon k. Pour chacun de ces composants, un

facteur d'échantillonnage relatif au format 4:2:0 existe et est parcouru selon *l* et *m*. C'est dans cette cinquième boucle imbriquée que se fait l'envoi des pixels vers le module *M_IHUFFMAN* (avec *comm_sendData*). L'envoi est suivi d'un *wait()* qui génère automatiquement un changement de contexte. Le module *M_PROC* reprendra son exécution lorsque *M_IHUFFMAN* aura terminé son exécution. L'exécution reprendra à la suite du *wait()* et tous les index de boucle seront restaurés à leur dernier état.

```
while(1)
{
  for (i=1; i<=vertical_mcus; i++)
  {
    for (j=1; j<=horizontal_mcus;j++)
    {
      for (k=0; k<number_of_image_components_in_frame; k++)
      {
        for (l=0; l<vertical_sampling_factor; l++)
        {
          for (m=0; m<horizontal_sampling_factor; m++)
          {
            pTableSelector.write(k);
            comm_sendData(); // envoi des pixels
            wait();
          }
        }
      }
    }
  }
}
```

Figure 4.9 : Implantation du noyau du décodage avec SystemC

La figure 4.10 montre le noyau de la version Syslib non bloquante. Cette version plus complexe nécessite d'abord de conserver tous les index de boucles dans des membres de la classe (variables *m_**). Les boucles *for* doivent être remplacées par des constructions *if-else* qui nuisent à la compréhension du code. D'autres idées d'implantation plus compactes existent, mais compliquent davantage la compréhension.

```
top_of_code:
if (m_i<=vertical_mcus)
{
  if (m_j<=horizontal_mcus)
  {
    if (m_k <number_of_image_components_in_frame)
    {
      if (m_l<vertical_sampling_factor)
      {
        if (m_m<horizontal_sampling_factor)
        {
          pTableSelector.write(m_k);
          comm_sendData();
          m_m++;
        }
        else
        {
            m_m = 0;
            m_l++;
            goto top_of_code;
        }
      }
      else
      {
        m_l = 0;
        m_k++;
        goto top_of_code;
      }
    }
    else
    {
      m_k = 0;
      m_j++;
      goto top_of_code;
    }
  }
  else
  {
    m_j = 1;
    m_i++;
    goto top_of_code;
  }
}
else
{
  m_i = 1;
}
```

Figure 4.10 :Implantation du noyau du décodage avec Syslib

4.7. Résultats quantitatifs

Nous allons maintenant comparer la performance d'exécution des différents exemples présentés ci-dessus. Les tests de performance comparent le temps d'exécution entre Syslib et SystemC pour l'exécution globale et pour les communications. Des analyses seront effectuées au fur et à mesure, puis nous terminerons cette section par une analyse de la taille des différents fichiers.

4.7.1. Additionneur et contrôleur mémoire

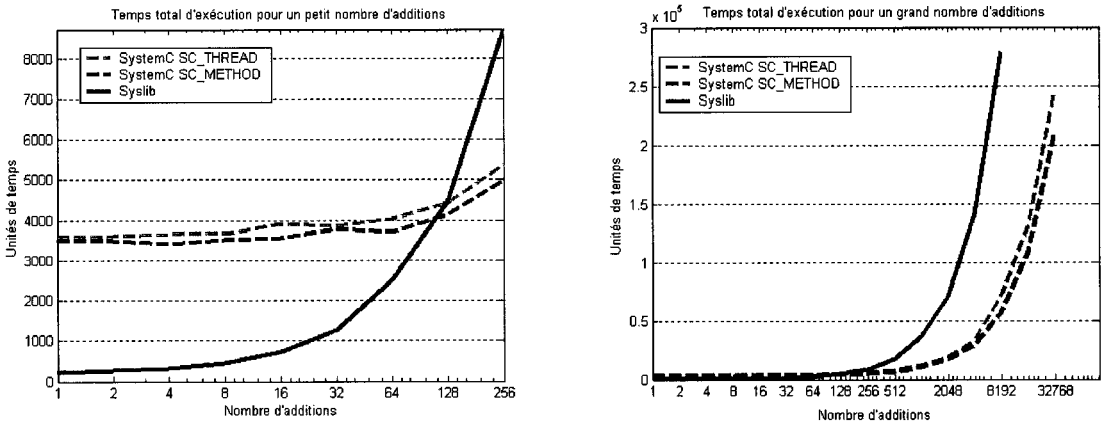
Nous observons pour ces deux exemples de design un comportement similaire menant à une analyse très intéressante. La méthodologie utilisée a été de simuler le système pour un nombre grandissant de requêtes. Dans le cas de l'additionneur, on exécutera le système pour un intervalle de 1 à 32768 additions aléatoires (en augmentant le nombre d'additions par puissance de 2). Les résultats comparent une implantation Syslib à une implantation SystemC utilisant des *SC_METHOD* et des *SC_THREAD*. Le tableau 4.1 affiche la moyenne du temps d'exécution total résultant. Nous rappelons que nous utilisons des unités de temps (*UT*) représentant le nombre d'incrément du compteur de performance Windows.

Tableau 4.1 : Temps d'exécution pour l'additionneur

Nombre d'additions	*SC_THREAD (UT)*	*SC_METHOD (UT)*	SYSLIB *(UT)*
1	3585,00	3501,00	226,67
2	3602,00	3488,67	265,67
4	3656,33	3399,33	331,33
8	3658,67	3495,33	456,00
16	3908,33	3557,33	724,00
32	3859,67	3784,67	1274,00
64	4047,67	3724,67	2520,67
128	4422,67	4146,00	4453,00
256	5361,00	4978,67	8724,67
512	7259,67	6575,33	17305,67
1024	11690,33	9975,67	36266,00
2048	19272,67	16952,00	70758,00
4096	34047,33	29463,67	141356,00
8192	71229,00	57813,00	279510,00
16384	131553,00	109022,33	-
32768	246051,33	212108,67	-

Les résultats sont présentés graphiquement à la figure 4.11. Nous voyons d'abord à la figure 4.11a que pour un petit nombre d'additions (1 à 16 additions), Syslib est plus performant que SystemC d'un facteur 10. Après, la performance diminue jusqu'à 128 là où SystemC et Syslib se rejoignent. Ensuite, lorsque le nombre d'additions est grand à la figure 4.11b, les performances de Syslib se détériorent énormément. Nous constatons

aussi que les *SC_METHOD* sont légèrement plus performants que les *SC_THREAD ;* phénomène attendu étant donné la structure du simulateur de SystemC [RHGK01].



(a) (b)

Figure 4.11 : Résultats pour l'additionneur

La méthodologie à suivre pour le contrôleur mémoire consiste à produire un nombre de requêtes mémoire aléatoires allant de 1 à 10 000 000. Le hasard génère soit une lecture ou une écriture de taille pouvant être un octet, un demi-mot ou un mot ayant pour destination soit la mémoire, soit le gestionnaire d'interruptions ou encore le générateur d'événements.

Tableau 4.2 : Temps d'exécution par requête pour le contrôleur mémoire

Nombre de requêtes	SystemC *(UT)*	Syslib *(UT)*
1	4189,67	366,33
10	430,83	121,73
100	54,41	77,30
1 000	17,41	73,42
10 000	13,62	73,16
100 000	13,40	72,58
1 000 000	13,26	72,49

Les résultats obtenus pour le contrôleur mémoire démontrent le même comportement que ceux obtenus avec l'additionneur. Nous les présentons cette fois au tableau 4.2 sous la forme de temps moyen par requête pour SystemC et pour Syslib.

Les résultats présentés graphiquement à la figure 4.12 démontrent clairement que pour un nombre de requêtes inférieur à 100, Syslib est légèrement plus performant. En augmentant le nombre de requêtes, le temps moyen pour une requête SystemC stagne autour de 13,5 UT alors qu'il converge autour de 72,5 UT avec Syslib, indiquant que SystemC est 5 fois plus performant. Une analyse fournie à la section 4.7.5 permet de comprendre ces résultats.

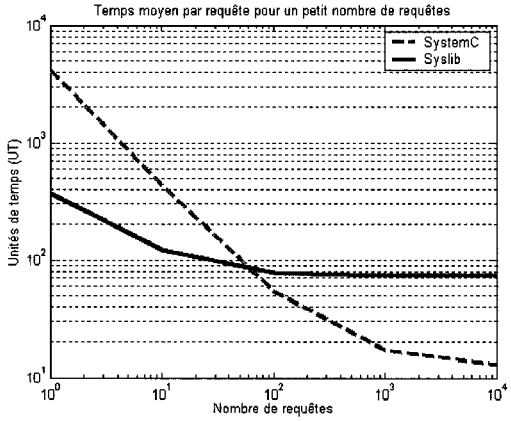


Figure 4.12 : Résultats pour le contrôleur mémoire

4.7.2. Décodeur JPEG

Les résultats présentés dans cette section compareront les performances de la première version du décodeur JPEG[7]. L'exécution de cette version du décodeur montre des résultats forts différents que ceux obtenus pour les deux exemples précédents. En effet, le décodeur JPEG est plus performant en Syslib qu'en SystemC. Les résultats importants

[7] Les résultats de la deuxième version du décodeur JPEG sont similaires à la première version.

sont présentés au tableau 4.3 et tracés à la figure 4.13. D'autres résultats pour le décodeur JPEG sont présentés en Annexe B.

Tableau 4.3 : Temps d'exécution pour le décodeur JPEG

Nom du fichier	Temps d'exécution (*UT*)			Écart de performance	
	SYSTEMC	SYSLIB avec hiérarchie	SYSLIB sans hiérarchie	écart pour Syslib sans hiérarchie	écart pour Syslib avec hiérarchie
128x128.jpg	268627	235927	170836	57.24%	13.86%
320x240.jpg	1240823	1096788	795922	55.90%	13.13%
384x480.jpg	2891514	2574706	1871207	54.53%	12.30%
400x592.jpg	3758112	3354695	2573860	46.01%	12.03%
416x608.jpg	4041546	3575819	2735027	47.77%	13.02%

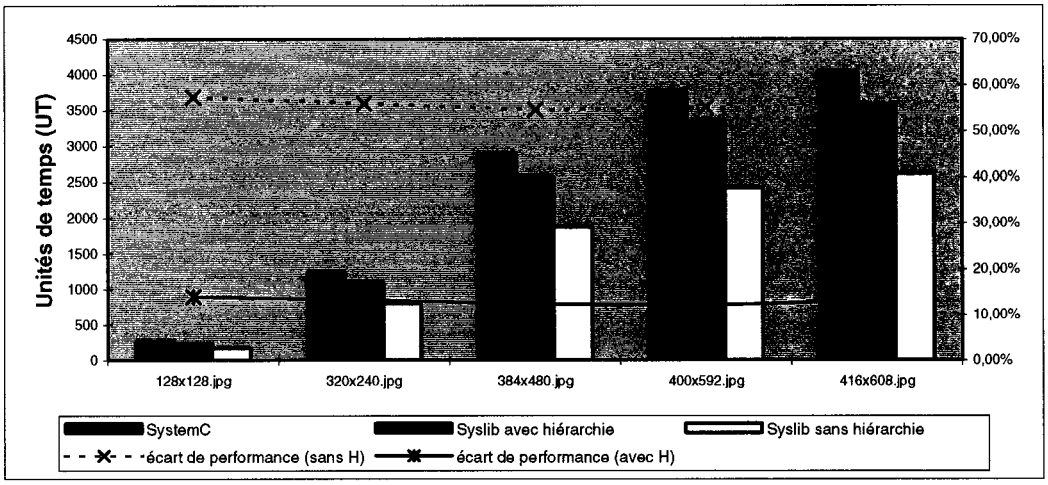


Figure 4.13 : Résultats pour le décodeur JPEG

Les résultats s'expliquent d'après les opérations de communication les plus fréquemment utilisées. Les lectures et les écritures de données sont présentes dans un ratio de 128:1 par rapport aux notifications d'événements. L'annexe A prouve que les lectures et écritures de données sont plus performantes en Syslib qu'en SystemC, d'où le gain de performance du décodeur JPEG en Syslib, d'environ 55% pour la version non hiérarchique et de 13% pour la version hiérarchique. À cause de son implantation, la version hiérarchique requiert un temps de traitement supplémentaire pour propager les données à tous les niveaux. Une analyse statique de la structure du système conçu aurait pu être effectuée au moment de la compilation pour une meilleure performance d'exécution.

Analyse des communications

Les options de débogage de la bibliothèque ont été utilisées pour démontrer leur utilité dans l'analyse des communications d'un système. Ces options permettent d'obtenir le temps d'exécution et le nombre total d'opérations de communications effectuées. Les résultats peuvent être affichés en fin d'exécution grâce à la méthode du simulateur *SysPrintTimings*. L'exécution du décodeur JPEG pour la décompression du fichier *320x240.jpg* nous donne le rapport des temps d'exécution présenté au tableau 4.4.

Tableau 4.4 : Rapport de temps d'exécution pour le décodeur JPEG

Opération	Occurrence	Temps total (*UT*)	Temps moyen par opération (*UT*)[8]
Lecture de donnée (*read*)	1 848 495	6 349 556	3,5
Écriture de donnée (*write*)	1 846 439	11 829 313	6,5
Notification d'événement (*notify*)	19 806	1 042 573	52,64
Lecture d'événement (*peek*)	18 000	67 855	3,75
Consommation d'événement (*consume*)	19 806	73 815	3,75
Ordonnancement seulement (pendant la notification)	19 806	824 422	41,62

Ce tableau nous renseigne grandement sur le temps passé pour les différentes opérations de communication. Les lectures et les écritures de données, la lecture d'événements et la consommation d'événements prennent un temps comparable puisque l'opération est du même ordre de complexité au niveau du simulateur. On constate par ailleurs que les notifications prennent un temps énorme comparativement aux autres opérations. Le rapport des temps d'exécution nous apprend également le temps nécessaire à l'ordonnancement même du module : ce temps d'ordonnancement est inclus dans le temps de notification. Nous concluons donc que le temps de l'ordonnancement est extrêmement grand par rapport aux autres opérations et que cela influence la performance des simulations. En SystemC, l'ordonnancement des modules est rendu possible grâce

[8] Certains résultats ce cette colonne sont approximatifs et peu précis, puisque la résolution du compteur de performance exerce une influence qu'on ne peut mesurer.

aux écritures simples, ce qui est beaucoup plus performant qu'en Syslib (avec les notifications). Dans ce cas, pour quelle raison la simulation est-elle plus rapide en Syslib qu'en SystemC? Simplement parce que le temps pour la notification est minime dans la simulation totale. Si l'on consulte la charte représentant le pourcentage du temps passé par catégorie d'opérations à la figure 4.14, on y constate que la notification représente environ 6% du temps d'exécution, ce qui est négligeable par rapport aux autres opérations plus rapides en Syslib qu'en SystemC. On remarque par ailleurs qu'environ 50% du temps total d'exécution est dépensé au niveau des communications, ce qui est un pourcentage important.

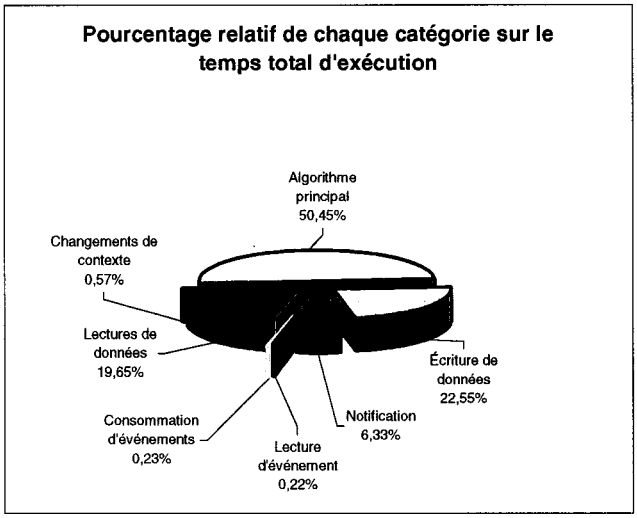


Figure 4.14 : Poids par catégorie pour le décodeur JPEG en Syslib

4.7.3. Routeur de paquets et *BlockMatcher*

Le *PacketRouter* présente des résultats similaires au décodeur JPEG sans hiérarchie de modules. L'analyse expliquant ce fait stipule que l'utilisation intensive de modules perpétuels (50% le sont dans cet exemple) diminue le temps relié à la notification puisque les modules se réordonnancent automatiquement.

Les résultats obtenus pour le *BlockMatcher* sont pratiquement similaires pour les implantations SystemC et Syslib : la version SystemC est généralement plus performante

d'environ 3% et ce pour les deux implantations (tableau de ports et FIFO). Il importe cependant de rappeler que la version de bibliothèque supportant la hiérarchie de modules est environ 40% moins performante que la version non hiérarchique. Par ces faits, nous pouvons affirmer qu'un version non hiérarchique du *BlockMatcher* serait plus performante que SystemC.

4.7.4. Taille des fichiers

Un argument de poids constitue la taille des fichiers générés pour les bibliothèques ainsi que pour les applications mêmes. Le tableau 4.5 montre ces résultats pour Syslib et SystemC. Nous voyons dans ce tableau que les fichiers Syslib sont plus petits d'un facteur d'environ 2,5. Syslib propose dans sa méthodologie d'embarquer les modules partitionnés en logiciel sur une ROM, d'où l'intérêt d'avoir de petits fichiers. En SystemC, on n'intégrera sans doute pas les fichiers tels quels, ce qui donne un travail supplémentaire pour changer de bibliothèque à ce niveau. Avec Syslib, la bibliothèque sera directement réutilisée pour le logiciel embarqué, avec certains ajouts et modifications.

Tableau 4.5 : Taille des fichiers pour les exemples présentés

Application	Taille pour Syslib (ko) *(avec hiérarchie)*	Taille pour SystemC (ko)
Bibliothèque de simulation	47 *(50)*	3060
Routeur de paquets	109	224
BlockMatcher	145	204
Décodeur JPEG	105	240
Contrôleur mémoire	69	212
Additionneur	69	196

4.7.5. Analyse des résultats

Des examens approfondis des résultats nous montrent que les performances du modèle peuvent être exprimées à l'aide d'une expression mathématique simple de la forme suivante :

$$T_{exécution} = T_{algorithmes} + S + \alpha \cdot T_{communications}$$

où

- $T_{exécution}$ est le temps total mesuré de l'exécution ;
- $T_{algorithmes}$ est le temps d'exécution des algorithmes du système ;
- S est le surcoût d'initialisation de l'engin de simulation ;
- α est le nombre d'opérations de communication ;
- $T_{communications}$ est le temps nécessaire par type de communications.

Cette équation est valide pour les temps d'exécution mesurés en SystemC et en Syslib. En posant d'abord l'hypothèse que $T_{algorithmes}$ est le même pour les deux bibliothèques, nous pouvons conséquemment l'exclure de l'analyse. Ensuite, nous pouvons détailler $\alpha T_{communications}$ comme étant $\alpha_1 T_{lecture} + \alpha_2 T_{écriture} + \alpha_3 T_{consommation} + \alpha_4 T_{notification}$ dans le cas de Syslib et $\alpha_1 T_{lecture} + \alpha_2 T_{écriture}$ dans le cas de SystemC, où les notifications sont incluses dans les écritures. À l'annexe A, le tableau A.1 nous fait remarquer que le temps pour une lecture ou une écriture est de même ordre avec Syslib et SystemC. Le tableau 4.4 nous indique par ailleurs que la notification d'événements Syslib est plus lente (d'un facteur 8) que l'écriture SystemC utilisée pour la notification.

Pour l'évaluation de S nous utiliserons l'exemple de l'additionneur et le tableau 4.1 qui nous apportent la plus juste valeur du surcoût de l'initialisation du simulateur. D'après ce tableau, nous pouvons affirmer que le temps moyen requis pour une addition est d'environ 34 UT pour Syslib et de 6,5 UT pour SystemC (mesuré lorsque le temps/addition converge). Ce temps peut être soustrait du temps total de simulation pour une seule addition, ce qui nous donne un surcoût d'initialisation d'environ 200 UT pour Syslib et de 3500 UT pour SystemC. Comme le surcoût d'initialisation relié au simulateur est très élevé pour SystemC, un grand nombre d'opérations est requis pour que ce surcoût devienne négligeable par rapport au temps total. Syslib sera ainsi avantagé pour les simulations plus courtes alors que SystemC le sera pour les simulations plus longues. Syslib sera également désavantagé lors des simulations où le nombre de notification est grand par rapport au nombre de lectures et d'écritures de données.

Performance des simulateurs Syslib et SystemC

Nous avons démontré au tableau 4.4 que le temps pour la notification est extrêmement lent. Comme toutes les opérations d'ordonnancement en Syslib doivent être effectuées par une notification, il s'ensuit une perte de performance dramatique par rapport à SystemC. L'ordonnancement en SystemC est *multi-thread* (donc en concurrence avec l'exécution de d'autres modules) et est donc fortement complexe à mesurer. En Syslib, il se fait pendant la notification, sur un *thread* unique. De plus, avec SystemC, le simulateur possède un caractère distinctif pour l'ordonnancement [RHGK01]. La première étape consiste à exécuter tous les processus ordonnancés. La seconde étape consiste en la propagation des événements pour réveiller tous les processus affectés par les changements de signaux. SystemC ne parcoure sa liste de processus à ordonnancer qu'une seule fois (à la deuxième étape). Les spécifications en SystemC UTF ne possèdent pas d'horloge, ce qui confine le travail du simulateur à ces deux étapes simples. À l'opposé avec Syslib, on parcoure l'ensemble (*map*) de modules à ordonnancer à chaque notification, ce qui explique en partie la perte de performance. Cette implantation peu efficace est cependant rapide à programmer et simple à comprendre. Il n'est pas possible de faire une analyse statique de l'ordre d'exécution des modules pour réduire le temps du simulateur, mais par contre, une analyse de la structure du système en début de simulation aurait permis de réduire les temps de recherche dans les structures de données pendant la simulation. La section 4.9 explique aussi un autre problème de la mauvaise performance de l'engin de simulation Syslib.

4.8. Résultats qualitatifs

Plusieurs détails de l'implantation ne se mesurent pas en chiffre, mais plutôt en mots. Également, les résultats quantitatifs nous forcent à faire des choix qui se répercuteront sur la qualité du code généré. Ce sont à ces questions et à d'autres que cette section tentera de répondre.

4.8.1. Considérations syntaxiques

Nous verrons ci-dessous des éléments qui influencent la syntaxe du code produit.

Représentation des éléments

Les deux bibliothèques sont toutes aussi simples à apprendre et à utiliser. Dans le cas de SystemC, on représente la classe par un *SC_MODULE* et le constructeur par un *SC_CTOR* qui sont en fait des *macros* qui cachent la classe et le constructeur du module. Certains vont sans doute préférer cette approche à celle de Syslib qui ne cache ni la classe, ni le constructeur. Comme le travail de conception de systèmes à haut niveau sera sans doute attribué à un concepteur logiciel, le recours à ces *macros* n'apporte pas de réels avantages pour un programmeur expérimenté.

Association des ports aux canaux

La façon de lier les ports avec Syslib utilise un concept simple, malheureusement lié à une syntaxe complexe. La méthode *bind* pour lier le port *pOut* du module *mMod1* au port *pIn* du module *mMod2* par le canal *cCanal* est utilisée de cette façon:

```
cCanal.bind((SysModule*)&mMod1, &mMod1.pOut, (SysModule*)&mMod2R, &mMod2.pIn);
```

comparativement à celle-ci pour SystemC :

```
mMod1.pOut(cCanal);
mMod2.pIn(cCanal);
```

ce qui est visuellement plus simple à lire. Des améliorations pourraient être effectuées à ce niveau.

Ports hiérarchiques

Dans les cas où il y a des modules hiérarchiques dans le système à coder, Syslib demande au programmeur de bien identifier les ports hiérarchiques afin que ceux-ci puissent être connectés aux modules externes. L'identification se fait ainsi par l'opération:

```
pPort.setHierarchicalFlag(true);
```

Cette opération n'existe pas en SystemC. En Syslib, l'usager doit entrer du code supplémentaire pour cette identification. Cependant, on doit garder à l'esprit que cette

opération pourra être générée automatiquement par l'outil Picasso (voir l'Annexe E). Une amélioration devrait être effectuée pour retirer cette ligne de code.

Modules hiérarchiques

Les modules hiérarchiques sont moins performants à cause d'une structure de données STL utilisée: une liste de *SysConnect*. Or, cette liste est nécessaire pour faire du multi-port (par exemple pour connecter une sortie à plusieurs entrées). Une solution pour palier à cette perte de performance est de faire un tableau de ports si on ne veut pas utiliser la hiérarchie de modules.

Densité de code

En prenant en considération tous les exemples présentés, nous remarquons que le code SystemC peut décrire les systèmes avec environ 10% moins de lignes de code. Les lignes de code supplémentaires sont principalement dues à l'ajout de fonctions pour la consommation d'événements et de données ainsi que pour le support structurel tel que présenté au chapitre 3 (section 3.1).

4.8.2. Styles de programmation

Pour mieux comprendre la structure et les objets utilisés lors de la construction de systèmes, il est recommandé de respecter un style de programmation basé sur le C++. Les types de données abstraits devraient toujours être utilisés pour tirer avantage des outils automatisés de transformation de code.

Nous proposons d'identifier les modules, ports et canaux tel que présenté dans le tableau 4.6. Bien souvent, les ports de différents modules auront des noms similaires, de même que le canal sur lequel l'information transitera. Il peut y avoir ambiguïté si deux modules nommés *module1* et *module2* possèdent chacun un port nommé *valeur* et sont reliés par un canal nommé *valeur*. L'utilisation de préfixes pour le nom des instances permet de différencier les types d'objets pour ainsi clarifier la construction des systèmes en design.

Tableau 4.6 : Style de programmation avec Syslib

Élément	Préfixe suggéré à l'objet créé	Exemple
SysModule	m	`SysModule mModule1;`
SysInt, SysUInt, etc.	d	`SysInt dValeur;`
SysEvent	e	`SysEvent eEvent;`
SysInPort, SysOutPort	p	`SysInPort<SysEvent> peMyPort;`
SysChannel	c	`SysChannel<SysChar> cdMyChannel;`

4.9. Améliorations

Un problème de performance flagrant est apparu pendant les premiers tests de la bibliothèque Syslib. Les communications au niveau des données (lectures et écritures) étaient beaucoup moins performantes que celles de SystemC. Une recherche nous a permis de constater que les performances de la bibliothèque *STL* utilisée pour les structures de données étaient très décevantes. SystemC utilise quant à elle des structures de données adaptées et optimisées.

Les structures de données STL implantant les FIFO sont des *queues*. Les méthodes utilisées pour cette structure sont *push* pour insérer un élément, *pop* pour en retirer un ainsi que *size* pour connaître le nombre d'éléments dans la file. Les références sur STL ont permis d'apprendre que l'espace alloué pour les *queues* est ajusté dynamiquement selon le nombre d'éléments dans la file, ce qui n'est pas utile pour Syslib qui fixe la taille de sa file avant le début de la simulation. Par ailleurs, la fonction *size* est de complexité $O(n)$ puisqu'elle compte le nombre d'éléments en file au lieu d'utiliser un membre ajusté selon la taille de la liste. Ces faits nous ont mené à implanter une version maison de la file d'attente. Les gains en performance ont été immédiatement constatés, tel qu'indiqué au tableau 4.7. De plus, nous avons amélioré la lecture Syslib d'un autre ordre en proposant une méthode qui permet d'affecter directement la valeur lue du canal à une variable temporaire dont l'adresse est fournie en paramètre. Il va de soi que les files STL ont été retirées de la version de Syslib et remplacées par ces structures qui représentent les meilleurs résultats.

Tableau 4.7 : Temps moyen mesuré pour les méthodes implantées avec STL

Temps moyen des méthodes	Temps moyen mesuré (*UT*)
Écriture dans une file STL	4,429
Écriture dans une file STL sans l'utilisation de la méthode size()	4,062
Écriture sans STL (tableau et pointeurs)	1,196
Lecture dans une file STL	5,710
Lecture dans une file STL sans l'utilisation de la méthode size()	4,806
Lecture sans STL (tableau et pointeur)	1,480
Lecture sans STL (tableau + pointeur + paramètre)	0,851

Le même problème existe au niveau des structures de données utilisées pour l'engin de simulation. Cette fois, une combinaison des *maps* et des *lists* est utilisée et leur performance est encore une fois décevante. L'implantation de structures de données adaptées est nécessaire mais complexe et demeure donc en suspens. En bref, l'utilisation de STL permet une implantation rapide de structures de données complexes et efficaces. Le sacrifice à fournir est celui de la performance obtenue.

Conclusion et travaux futurs

Après avoir passé en revue les langages et les méthodes de conception de systèmes embarqués, nous avons proposé une méthodologie de codesign intégrant la bibliothèque Syslib. Il a été décidé que pour ce projet, Syslib ne couvrirait pas tous les aspects de la méthodologie proposée, mais seulement les aspects fonctionnels sans notion de temps. Les systèmes modélisés à l'aide de la bibliothèque utilisent des notions structurelles de modules, ports, canaux, événements et données. Syslib offre des services tels la hiérarchie de modules et la concurrence. Syslib propose des concepts de modélisation intéressants qui sont peu exploités dans les autres méthodologies, entre autre les modules possédant de multiples unités d'exécution, les unités d'exécution perpétuelles, les ports ayant une valeur persistante et la modélisation non bloquante. Syslib représente également un niveau système pour Cynlib, bibliothèque de modélisation de matériel. Cynlib ne propose aucune solution en ce genre ce qui fait de Syslib un choix intéressant.

L'utilisation de l'orienté objet pour la modélisation des systèmes est une solution flexible et intelligente, qui permet d'emprunter une approche de conception par raffinement progressif. Nous avons expliqué que les systèmes utilisant le même langage de programmation à tous les niveaux d'abstraction peuvent être plus efficacement raffinés.

Pour valider les fonctionnalités de Syslib, plusieurs exemples de design ont été programmés. Par ces multiples exemples, dont le routeur de paquets, le décodeur JPEG et le contrôleur mémoire, nous avons pu constater les possibilités mais aussi les limites de Syslib. Nous avons démontré que les modules programmés avec Syslib sont orientés pour une implantation en matériel. Les multiples comparaisons entre Syslib et SystemC, le standard *de facto* de l'industrie, ont permis de démontrer le potentiel de Syslib. Pour certains exemples, Syslib s'est montré plus performant que SystemC, notamment pour les applications orientées vers les données.

Il se pourrait que les concepts et les modèles de programmation proposés ne répondent pas exactement aux besoins d'une application. Par exemple, l'orientation très logicielle de SystemC de niveaux UTF et TF met en doute la simplicité d'un raffinement éventuel vers le matériel, puisque d'un niveau à l'autre, les spécifications sont très éloignées. Comme SystemC est un engin de simulation lourd et orienté pour le matériel, il y a fort à parier que pour un partitionnement logiciel, des spécifications en SystemC soient portées vers une bibliothèque plus légère, plus simple et possédant des services de RTOS. Dans le cas de Syslib, on constate que les spécifications seront directement réutilisées pour un partitionnement logiciel (on ne change que la bibliothèque derrière, l'API reste le même). Pour le partitionnement en matériel vers Cynlib, des changements majeures doivent prendre place, mais la structure des programmes est déjà orientée pour le matériel.

Syslib sert d'intermédiaire entre des spécifications codées en C++ pur et des spécifications raffinées vers une architecture précise. On n'aura qu'à penser que Syslib offre des services de construction spécialisés pour les systèmes embarqués. Au même titre, il aurait été possible de développer un système à niveau d'abstraction élevé en utilisant les services d'une bibliothèque de *threads* telle POSIX. Ceci lance donc un doute sur l'utilité d'une bibliothèque de modélisation de niveau UTF ou TF. Dans les cas où on voudra utiliser SystemC pour modéliser du matériel, l'utilisation des niveaux UTF et TF pour les spécifications abstraites est justifiable. Autrement, SystemC est une solution comme une autre. Dans le cas de Syslib, son utilisation est justifiable pour n'importe quelle bibliothèque matérielle sous-jacente (SystemC ou Cynlib).

Forte Design Systems ne planifie plus de continuer le développement de Cynlib mais de s'orienter plutôt vers la bibliothèque SystemC. C'est pourquoi les niveaux RTL et BCA de SystemC devraient être pris en considération pour les futures possibilités de Syslib. Un système pourrait très bien être composé d'un mélange de spécifications codées en Syslib et en SystemC. C'est pourquoi la méthodologie proposée dans ce mémoire a été modifiée pour tenir compte de cette nouvelle bibliothèque. Cette méthodologie baptisée

« méthodologie de codesign du CIRCUS » est présentée à la figure 4. Elle intègre Syslib ou SystemC au niveau de l'entrée des spécifications. On propose de raffiner la spécification vers SystemC BCA pour les décisions de partitionnement. Une plate-forme d'exécution SystemC en développement permettra de mieux orienter ces décisions. Pour le partitionnement vers le matériel, SystemC et Cynlib cohabitent, par le biais de la nouvelle bibliothèque ESC de Forte Design Systems. Dans le cas d'un partitionnement vers le logiciel, on propose toujours de conserver l'interface Syslib^{LL} qui lie le RTOS embarqué µC/OS-II aux spécifications logicielles. Tous les niveaux de la méthodologie sont maintenant supportés par la vérification fonctionnelle et l'outil graphique Picasso.

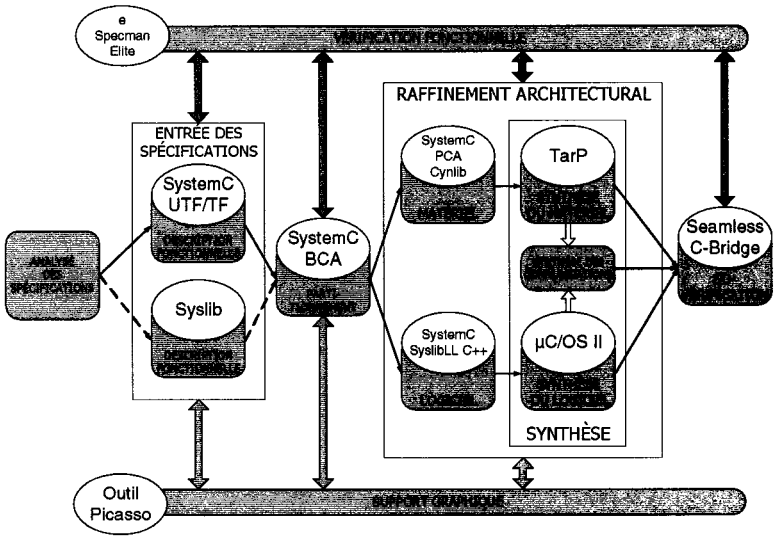


Figure 4 : Méthodologie de codesign du CIRCUS

Plusieurs possibilités viennent en tête quant à la possibilité de simuler un système composé de modules programmés en Syslib et en SystemC. La première consiste à exécuter les deux simulateurs en parallèle et à tenter de trouver des techniques d'interopérabilité pour lier les modules, ports, canaux des bibliothèques. Une autre serait simplement de transformer le simulateur Syslib pour qu'il supporte des modules SystemC. Également, il y aurait moyen de faire une transformation syntaxique des spécifications Syslib vers SystemC, en analysant tous les modèles de programmation

possibles puis en trouvant des équivalents SystemC. Une base pour cette étude serait sans doute ESC qui lie Cynlib et SystemC.

Les travaux qui pourraient découler de ce projet sont nombreux. On pourrait croire que la nécessité de créer un niveau comportemental de spécifications soit mise en doute, notamment à cause de la venue de SystemC. Or, beaucoup de services proposés à l'origine pour Syslib^{BL} ne sont pas offerts dans SystemC, comme le choix du mécanisme d'ordonnancement des *threads*, la priorité des *threads* ou même les contraintes de temps. Ces ajouts pourraient être intégrés directement à SystemC ou encore un nouveau niveau Syslib pourrait être créé. Le niveau Syslib^{LL} garde toute son importance puisque peu de recherches existent dans ce domaine, comme si on prenait pour acquis l'intégration de logiciel sur les microprocesseurs embarqués. Beaucoup de travail reste à faire quant à l'automatisation de la génération d'interfaces pour intégrer plus facilement le logiciel à une architecture ciblée. L'orientation vers un RTOS générique pourrait également être étudiée.

Dans une autre optique, plusieurs bibliothèques, dont SpecC, incluent des composants fonctionnels à même les bibliothèques. Ces composants se retrouvent typiquement sur une plate-forme d'exécution. Il s'agit de mémoires et de registres configurables, de bus et d'arbitres, de protocoles d'échange de données ou encore de modules plus complexes, comme des générateurs d'événements et de nombres aléatoires ou encore des interfaces de débogages. Tous ces composants pourraient être programmés avec la bibliothèque Syslib (et même Cynlib où nous l'avons fait avec le *CynFifo*) puis être offert dans une banque de composants. Ces modules supplémentaires devraient refléter l'architecture sous-jacente ciblée (TarP).

Références

[Acce02] Accellera 2002. Page du groupe. [En ligne]. http://www.accellera.org/ (Page consultée le 31 septembre 2002)

[AKB00] ALEXANDER Perry, KAMATH Roshan, BARTON David. 2000. «System Specification in Rosetta». *7th IEEE International Conference and Workshop on Engineering and Computer Based Systems*. Edinburgh, Royaume-Uni. 299-307.

[Alex01] ALEXANDRESCU Andrei. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Boston : Addison Wesley. 352p.

[ARM00a] ARM. 2000. *ARM7TDMI Data Sheet*. [En ligne]. Cambridge, Angleterre: Advanced RISC Machines. 268p. ARM DDI 0029E. http://www.arm.com /arm/documentation?OpenDocument (Page consultée le 10 novembre 2001)

[ARM00b] ARM. 2000. *AMBA – Advanced Microcontroller Bus Architecture Specification*. [En ligne]. Cambridge, Angleterre: Advanced RISC Machines. ARM IHI 0011A. http://www.arm.com/arm/ documentation?OpenDocument. (Page consultée le 10 novembre 2001)

[ArMa99] ARPNIKANONDT Chonlameth, MADISETTI Vijay K. 1999. *Constraint-Based Codesign (CBC) of Embedded Systems: The UML Approach*. Atlanta, GA : Georgia Tech. 19p. YES-TR-99-01.

[BaGa01] BAILEY Brian, Gajski Daniel. 2001. «RTL Semantics and Methodology». *14th International Symposium on System Synthesis*. New York : ACM. 69-74.

[BaOb99] BAKER Mark, O'BRIEN-STRAIN Eammon. 1999. «Co-Design Made Real: Generating and Verifying Complete System Hardware and Software Implementations». *Embedded Systems Conferences*. [En ligne].

http://www.esconline.com/99fallpapers.htm (Page consultée le 4 novembre 2002)

[BCGH97] BALARIN Felice, CHIODO Massimiliano, GIUSTO Paolo, HSIEH Harry et al. 1997. *Hardware-Software Co-design of Embedded Systems – The POLIS Approach* . Norwell, MA : Kluwer Academic Publishers. 297p.

[BeFi01] BERTOLA Marc, FILION Luc, BOIS Guy. 2001. *Block Matcher Seamless CVE Tutorial.* [En ligne] 38p. http://www.grm.polymtl.ca /circus/commun/publications/ (Page consultée le 4 novembre 2002)

[BeHo89] BELINA Ferenc, HOGREFE Dieter. 1989. «The CCITT-Specification and Description Language SDL». *Computer Networks and ISDN Systems.* B.V. Hollande. 16. 311-241.

[BeLe00] BERGAMASHI R.A., LEE W.R. 2000. «Designing System-On-Chip using cores». *Proceedings of the IEEE Design Automation Conference.* [En ligne]. http://www.dac.com (Page consultée le 5 août 2002)

[Berg00] BERGERON Janick. 2000. *Writing Tesbenches – Functional Verification of HDL Models.* Norwell, MA : Kluwer Academic Publishers. 1-19.

[Bert03] BERTOLA Marc. 2003. *Conception, réalisation et étude d'une plate-forme générique basée sur le protocole AMBA AHB.* Mémoire de maîtrise, Département de génie électrique, École Polytechnique de Montréal.

[BRJ99] BOOCH G., RUMBAUGH J., JACOBSON I. 1999. *The Unified Modeling Language User Guide*, Boston: Addison-Wesley. 512p.

[BuWe97] BURNS Alan, WELLINGS Andy. 1997. *Real-Time Systems and Programming Languages* 2^e *édition.* Boston: Addison-Wesley Longman. 2-15.

[CCHM99] CHANG Henry, COOKE Larry, HUNT Merrill, MARTIN Grant et al. 1999. *Surviving the SoC Revolution – A Guide to Platform-Based Design.* Norwell, MA : Kluwer Academin Publishers. 13,223-224.

[CIRC02] CIRCUS. 2002. Groupe de recherche sur le codesign de l'École polytechnique de Montréal. [En ligne]. http://www.grm.polymtl.ca/circus (Page consultée le 4 novembre 2002)

[Cowa02] COWARE. 2002. Coware N2C System Designer. [En ligne]. http://www.coware.com (Page consultée le 15 octobre 2002)

[Cyr01] CYR Geneviève. 2001. *Développement d'une interface matérielle configurable pour un processeur ARM7 basée sur le protocole VCI de VSIA*. Mémoire de maîtrise, Département de génie électrique, École Polytechnique de Montréal. 76-88.

[DaCl02] DART Sean, CLINE Brett. 2002. «Design & Methodology – 1 : Convergence of Design and Verification». EDA Vision : The Online Magazine for EDA Professionals. [En ligne]. 2:7. http://www.edavision.com (Page consultée en août 2002)

[Davi00] DAVIS, M. 2000. *Unicode Regular Expression Guidelines v5.1*. [En ligne]. Unicode Technical Report #18. http://www.unicode.org/unicode/reports/tr18/ (Page consultée le 5 juin 2002)

[DeDe98] DEITEL H.M., DEITEL P.J. 1998. *C++ How to Program 2nd Edition – ANSI/ISO C++ Draft Standard Starring the Standard Template Library*. Upper Saddle River, NJ : Prentice-Halls. 1130p.

[DGG01] DÖMER Rainer, GERTSLAUER Andreas, GAJSKI Daniel. 2001. *SpecC Language Reference Manual Version 1.0*. Tokyo : SpecC Technology Open Consortium. 88p. [En ligne]. http://www.specc.org/ (Page consultée le 18 septembre 2002)

[DGOS01] DOUCET Frédéric, GUPTA Rajesh, OTSUKA Masato, SCHAUMONT Patrick et al. 2001. «Interoperability as a Design Issue un C++ Based Modeling Environments». *14th International Symposium on System Synthesis*. New York : ACM. 87-92.

[DHKL00] DAVIS John, HYLANDS Christopher, KIENHUIS Bart, LEE Edward A., et al. 2000. *Ptolemy II : Heterogeneous Concurrent Modeling and Design*

in Java (Document Version 2.0.1). [En ligne]. Berkeley : DEECS, University of California. UCB/ERL M02/23. http://ptolemy.eecs.berkeley.edu (Page consultée le 18 septembre 2002)

[Dold01] DOLDI Laurent. 2001. *SDL Illustrated - Visually design executable models*. Toulouse:, France: Laurent Doldi Publishing. 270p.

[Duml02] DUMLUGÖL Dündar. 2002. «HW/SW Co-Design with SystemC». *IEEE Electronic Design Processes Workshop*. [En ligne]. http://www.eda.org/edps/edp02/SLIDES/edp02-s1_3-slides.pdf (Page consultée le 30 septembre 2002)

[Elli99] ELLIOTT John P. 1999. *Understanding Behavioral Synthesis : A Practical Guide to High-Level Design*. Norwell, MA: Kluwer Academic Publishers. 5-40; 77-79; 245-262.

[FaKh99] FAYAD Ghassan, KHORDOC Karim. 1999. «Une approche orientée objet pour la co-conception de systèmes embarqués dans le domaine des applications multimédias». *68e Congrès de l'ACFAS.* [En ligne]. http://www.acfas.ca/congres/congres68/ (Page consultée le 8 août 2001)

[FCBA02] FILION Luc, CHEVALIER Jérôme, BOIS Guy, ABOULHAMID El Mostapha. 2002. «The Syslib-Picasso Methodology for the Co-Design Specification Capture Phase». *System-on-Chip for Real-Time Applications.* Norwell MA : Kluwer Academic Publishers. 185-194.

[FDS00a] Forte Design System. 2000. *Cynlib Language Reference Manual v1.4.* [En ligne]. 80p. http://www.forteds.com (Page consultée le 8 août 2002)

[FDS00b] Forte Design System. 2000. *Cynlib Users Manual v1.4.* [En ligne]. 88p. http://www.forteds.com (Page consultée le 8 août 2002)

[FDS02a] Forte Design Systems Press Releases. 2002. «Forte Design Systems' Perspective Results Analysis Products Adds Supports for OpenVera». [En ligne] http://www.forteds.com/company/news/NewsPerspectiveVera-04-08-02.html . (Page consultée le 14 mai 2002)

[FDS02b] Forte Design Systems. 2002. Page de la compagnie. [En ligne]. http://www.forteds.com (Page consultée le 2 novembre 2002)

[FiBA02a] FILION Luc, BOIS Guy, ABOULHAMID El Mostapha. 2002. « Picasso: une méthodologie complète pour la conception des systèmes embarqués». *70e congrès de l'ACFAS.* [En ligne]. http://www.acfas.ca/congres/congres70 (Page consultée le 17 juillet 2002)

[FiBA02b] FILION Luc, BOIS Guy, ABOULHAMID El Mostapha. 2002. « Syslib: A System-Level Library Extended from Cynlib for SoC». *Proceedings of the 11th International HDL Conference & Exhibit.* San Jose, CA : Randall Bilof Publishing. 191-197.

[FiBA02c] FILION Luc, BOIS Guy, ABOULHAMID El Mostapha. 2002. « Syslib: A Comparison of Two C++ Bibliothèques for SoC Design at an Untimed Level of Abstraction». Soumis à *2003 Design Automation and Test in Europe.* [En ligne]. http://www.grm.polymtl.ca/~filion/data/date03.pdf (Page consultée le 4 novembre 2002)

[FiHe01] FILION Luc, HENEAULT Yannick et al. 2001. *Picasso – Guide du programmeur.* [En ligne]. Groupe de recherche sur le codesign (CIRCUS). 17p. http://www.grm.polymtl.ca/circus/interne (Page consultée le 4 novembre 2002)

[Fili02] FILION Luc. 2002. *Syslib^{FL} - Manuel de l'usager, spécification théorique et technique de la bibliothèque.* [En ligne]. Groupe de recherche sur le codesign (CIRCUS). 39p. http://www.grm.polymtl.ca/circus/interne (Page consultée le 4 novembre 2002)

[Fili02b] FILION Luc. 2002. Code source des exemples de design Syslib et SystemC programmés dans le cadre du projet de M.Sc.A de Luc Filion. [En ligne]. http://www.grm.polymtl.ca/circus/fr/projets/syslib (Page créée pour référence le 4 novembre 2002).

[GaRa94] GAJSKI Daniel, RAMACHANDRAN Loganath. 1994. « Introduction to High-Level Synthesis». *IEEE Design & Test of Computers*, 11:4. 44-54.

[Gero00] GEROUSIS Vassilios. 2000. «ALC – Accellerra». Présentation à *Forum on Design Languages*. [En ligne]. http://ecsi.fzi.de/fdl2000/ (Page consultée le 16 septembre 2002)

[GLMS02] GRÖTKER Thorsten, LIAO Stan, MARTIN Grant, SWAN Stuart. 2002. *System Design with SystemC*. Boston: Kluwer Academic Publishers. 1-9; 41-48; 77-83; 131-132; 185-188.

[GoGB97] GONG Jie, GAJSKI Daniel, BAKSHI Smita. 1997. «Model Refinement for Hardware-Software Codesign». *ACM Transactions on Design Automation of Electronic Systems*. 2:1. 22-41.

[Gray98] GRAY John S. 1998. *Interprocess Communications in UNIX – The Nooks & Crannies 2nd Edition*. Upper Saddle River, NJ : Prentice Hall. 462p.

[GuHo93] GUTTAG J.V., HORNING J.J. 1993. *Larch: Languages and Tools for Formal Specification*. [En ligne]. New York: Springler-Verlag. 250p. http://www.sds.lcs.mit.edu/spd/larch/ (Page consultée le 4 novembre 2002)

[GVNG94] GAJSKI Daniel, VAHID Frank, NARAYAN Sanjiv, GONG Jie. 1994. *Specification and Design of Embedded Systems*. Upper Saddle River, NJ : Prentice Halls. 63-88.

[GZDG00] GAJSKI Daniel D., ZHU Jianwen, DÖMER Rainer, GERSTLAUER Andreas, et al. 2000. *SpecC: Specification Language and Methodology*. Norwell, MA : Kluwer Academic Publishers. 1-5; 14-22; 55-68.

[Händ02] HÄNDEL Lars. 2002. *The Function Pointer Tutorials*. [En ligne]. http://www.newty.de (Page consultée le 17 janvier 2002)

[HBAB99a] HÉNEAULT Yannick, BOIS Guy. ABOULHAMID El Mostapha, BAILLAIRGÉ, Jacques et al. 1999. «Picasso: A H/S Capture Tool Based on VSIA Recommendations». *Proceedings of International Workshop On IP Based Synthesis and System Design*. [En ligne]. http://www.grm.polymtl.ca/circus/data/ (Page consultée le 16 mars 2001)

[HBAB99b] HÉNEAULT Yannick, BOIS Guy, ABOULHAMID El Mostapha, BAILLAIRGÉ, Jacques, et al. 1999. « Renoir Extensions to Support a H/S

Co-Design Methodology» *Proceedings of 16th Annual Intern. Conference Mentor Graphics User's Group*. [En ligne]. http://www.grm.polymtl.ca/circus/data/ (Page consultée le 18 mars 2001)

[Hene01] HÉNEAULT Yannick. 2001. *BasicARM - Technical document for HW/SW codesign and simulation on the BasicARM Platform.* [En ligne]. Groupe de recherche sur le codesign (CIRCUS). 31p. http://www.grm.polymtl.ca/circus/interne.

[HePa96] HENNESSY John L., PATTERSON Davis A. 1996. *Architecture des ordinateurs – Une approche quantitative 2e édition.* Paris : Morgan Kaufmann Publishers. 3-5, 527-544.

[HFBA01] HÉNEAULT Y., FILION L., BOIS G., ABOULHAMID, E.M. 2001. «A Fast Hardware Co-Specification and Co-Simulation Methodology Integrated in a H/S Co-Design Platform» *13th International Conference on Microelectronics*. [En ligne]. http://www.grm.polymtl.ca/circus/data/ (Page consultée le 4 février 2001)

[HoLF02] HOLLOWAY Steve, LONG David, FITCH Alan. 2002. *From algorithm to SoC with SystemC and CoCentric System Studio.* San Jose, CA : Synopsys Users Group. 27p.

[IMEC02] IMEC. Page du groupe. [En ligne]. http://www.imec.be (Page consultée le 4 septembre 2002)

[Jack02] JACK Keith. 2002. *Video Demystified : A Handbook for the Digital Engineer 3rd Edition.* Burlington, MA : Butterworth-Heinemann Publishers. 784p. 15-34.

[Jerr97] JERRAYA Ahmed.A. et al. 1997. *Multilanguage Specification for System Design.* [En ligne]. Grenoble, France : TIMA/SLS. 34p. http://tima.imag.fr/sls/documents/asi.pdf

[Kahn74] KAHN Georges. 1974. «The Semantics of a simple language for parallel programming». In J.L. Rosenfeld, editor, *Information processing 74 : proceedings of IFIP.* North-Holland Publishing Company. 46-77.

[Klei00] KLEIN G. 2000. *JFlex – The Fast Lexical Analyser Generator for Java User's Manual v1.3.* [En ligne]. http://www.informatik.tu-muenchen.de/~kleing/jflex/index.html (Page consultée le 17 août 2001)

[Klei02] KLEIN Russ. 2002. *Hardware/Software Co-Simulation.* [En ligne]. Mentor Graphics Corporation White Paper. 6p. http://www.mentor.com/seamless/tpapers.cfm . (Page consultée le 1er octobre 2002)

[Labr99] LABROSSE J. J. 1999. *MicroC/Os-II. The Real-Time Kernel.* Lawrence, KS : R&D Books. 498p.

[Lee99] LEE Edward A. 1999. *Overview of the Ptolemy Project.* [En ligne]. Berkeley : DEECS University of California. 14p. UCB/ERL M98/72 http://ptolemy.eecs.berkeley.edu. (Page consultée le 7 septembre 2002)

[Lefr02] LEFRANÇOIS Hugo. 2002. *Transformation automatisée de code source.* 36p. Rapport de projet de fin d'études au baccalauréat, Département de génie informatique, École Polytechnique de Montréal.

[LSJH00] LENNARD Christopher K., SCHAUMONT Patrick, DE JONG Gjalt, HAVERINEN Anssi et al. 2000. «Standards for System-Level Design: Practical Reality or Solution in Search of a Question?». *Proceedings of the IEEE Design Automation Conference.* [En ligne]. http://www.dac.com (Page consultée le 3 avril 2001)

[Ment00] Mentor Graphics. 2000. *Seamless User's and Reference Manual. Software Version 4.0.* [Logiciel]. Documentation de l'outil.

[Ment02a] Mentor Graphics. 2002. *Platform Express User's Guide Release 1.1.* [En ligne]. http://www.mentor.com/soc/verification/ (Page consultée le 27 octobre 2002)

[Ment02b] Mentor Graphics. 2002. *Seamless C-Bridge Technology Enabling C in Hardware/Software Co-Verification Datasheet.* [En ligne]. http://www.mentor.com/seamless/datasheets/cbridge/ (Page consultée le 27 octobre 2002)

[Micr97] MICROSOFT. 1997. *HOWTO: Use QueryPerformanceCounter to Time Code.* [En ligne]. Knowledge Base Article Q172338. http://support.microsoft.com/default.aspx?;Q172338& (Page consultée le 20 juillet 2002)

[Micr02] MICROSOFT. 2002. *setjmp/longjmp* – Documentation MSDN de Visual C++ v6.0 [Logiciel].

[NVG92] NARAYAN S., VAHID F., GAJSKI D. 1992. «System Specification with the SpecCharts Language». *IEEE Design and Test of Computers.* 9:4. 6-13.

[OSCI02a] OSCI. 2002. *SystemC Version 2.0.1 User's Guide.* [En ligne]. http://www.systemc.org (Page consultée le 1er juin 2002)

[OSCI02b] OSCI. 2002. *Functional Specification for SystemC 2.0.1- Version 2.0-Q* [En ligne]. http://www.systemc.org (Page consultée le 1er juin 2002)

[PeMi93] PENNEBAKER William B., MITCHELL Joan L. 1993. *JPEG – Still Image Data Compression Standard.* New York : International Thomson Publishing. 29-41; 65-79; 97-126; 169-185.

[RHGK01] RUF J., HOFFAMNN D., GERLACH J., KROPF T., et al. «The Simulation Semantics of SystemC». *Proceedings of the IEEE Design Automation Conference.* [En ligne]. http://www.dac.com (Page consultée le 4 septembre 2002)

[RPS01] RASHINKAR Prakash, PATERSON Peter, SINGH Leena. 2001. *System-on-a-chip verification methodology and techniques.* Norwell, MA : Kluwer Academic Publishers. 7-13.

[Sant01] Santarini Michael, 2001. « Cynlib joins OSCI, merging system-level languages ». *EETimes.com.* [En ligne]. http://www.eetimes.com/story/OEG20011113S0058 (Page consultée en juillet 2001)

[Schi99] SCHIRRMEISTER Frank. 1999. *Integration Platform Based Design using Cierto™ Virtual Component Co-Design (VCC).* [En ligne]. Cadence VCC

White Paper and Overview for Customers. http://www.cadence.com (Page consultée le 15 septembre 2002)

[Schw02] SCHWARTZ Kurt. 2002. «Modeling with SystemC 2.0». *Tutorial at International HDL Conference & Exhibit in San Jose*. Document interne.

[Seli99] SELIC Bran, 1999. «Turning clockwise: using UML in the real-time domain». *Communications of the ACM*. 42:10. 46-54.

[Suth02] SUTHERLAND Stuart. 2002. «Verilog, The Next Generation: Accellera's SystemVerilog». *Special Session #4.4 at International HDL Conference & Exhibit in San Jose, CA*. Document interne.

[Swan01] SWAN Stuart. 2001. *An Introduction to System Level Modeling in SystemC 2.0*. [En ligne]. SystemC White Paper Page. http://www.systemc.org (Page consultée le 11 juin 2001)

[Syno02] Synopsys. 2002. *CoCentric System Studio Data Sheet*. [En ligne]. http://www.systems.com/products/cocentric_studio /cocentric_studio_ds.pdf (Page consultée le 29 octobre 2002)

[VaGi02] VAHID Frank, GIVARGIS Tony. 2002. *Embedded System Design – A Unified Hardware/Software Introduction*. New York : Wiley & Sons Inc. 1-17; 207-243.

[VCPD99] VERKEST Diederik, COCKX Johan, POTARGENT Freddy, DE JONG Gjalt et al. 1999. «On the use of C++ for system-on-chip design». *IEEE Computer Society Workshop on VLSI'99*. 42-27.

[VIKJ94] VOSS Markus, ISMAIL Tarek Ben, JERRAYA Ahmed A., KAPP Karl-Heinz. 1994. «Towards a Theory for Hardware/Software Codesign». *Proceedings for the International Hardware/Software Codesign*. 173-180.

[VKS00] VERKEST Diederik, KUNKEL Joachim, SCHIRRMEISTER Frank. 2000. «System Level Design Using C++». *Proceedings of Design, Automation and Test in Europe (DATE'00)*. 74-83.

[VSB99] VERNALDE S., SCHAUMONT P., BOLSENS I. 1999. «An Object Oriented Programming Approach for Hardware Design». *IEEE Computer Society Workshop on VLSI'99*. 68-75.

[VSIA00a] VSIA System-Level Design Development Working Group. 2000. *System-Level Interface Behavioral Documentation Standard (SLD 1 1.0).* Disponible aux membres de VSIA.

[VSIA00b] VSIA On-Chip Bus Development Working Group 2000. *Virtual Component Interface Specification (OCB 2 2.0).* Disponible aux membres de VSIA.

[VSIA01] VSI Alliance Functional Verification Working Group 2001. *Taxonomy of functional verification for virtual component development and integration.* Disponible aux membres de VSIA.

[VSIA02] VSIA Virtual Socket Interface Alliance. Page du groupe. [En ligne]: http://www.vsi.org/ (Page consultée le 17 octobre 2002)

[VSIA97] VSIA. 1997. *VSI Alliance Architecture Document v1.0.* Disponible aux membres de VSIA.

[VSVE01] VANMEERBEECK G., SCHAUMONT P., VERNALDE S., ENGELS M., et al. 2001. «Hardware / Software Partitioning of embedded system in OCAPI-xl». *Proceedings of the 9th International Symposium on Hardware/Software Codesign*. 30-35.

[Wall91] WALLACE Gregory K. 1991. «The JPEG Still Picture Compression Standard». *Communications of the ACM*. 34:4. 30-44.

[Watk98] WATKINSON John. 1998. *MPEG-2*. Burlington, MA : Butterworth-Heinemann Publishers. 160-221.

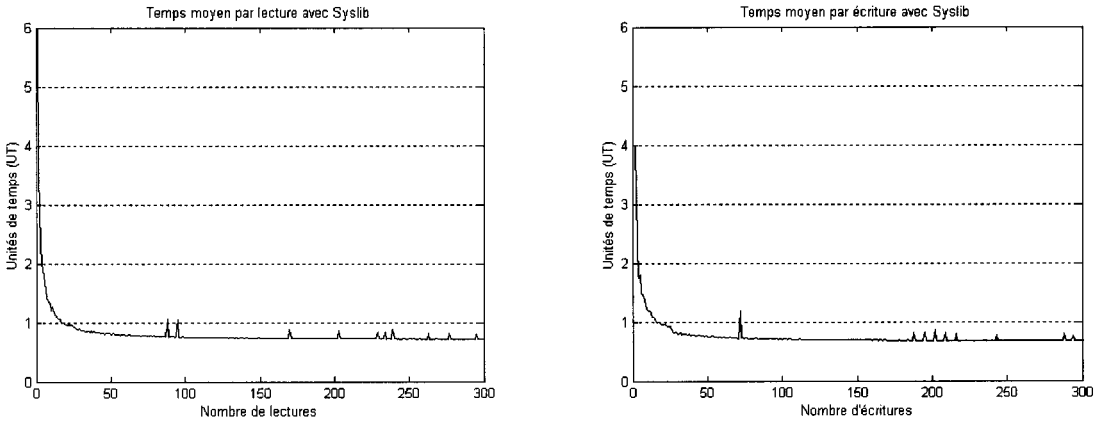
[Word92] Wordsworth J.B. 1992. *Software Development with Z – A Practical Approach to Formal Methods in Software Engineering*. Boston : Addison Wesley Publishers. 334p.

Annexes

ANNEXE A

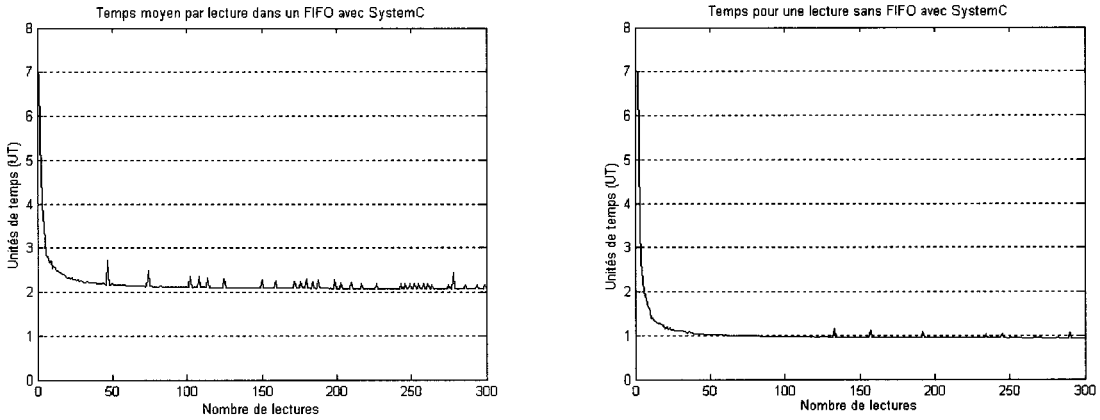
Performance des lectures et des écritures

Nous avons mesuré des temps pour des lectures et des écritures de données en Syslib et en SystemC. Pour ce faire, nous avons utilisé le compteur performant de Windows. Nous avons mesuré le temps des opérations (lectures/écritures) dans une boucle en incrémentant le nombre de boucles. Les temps sont illustrés aux figures A.1, A.2 et A.3 et représentent des moyennes. Nous observons sur ces figures que lorsque la boucle effectue un petit nombre d'opérations, le temps amputé par opération est élevé. Par contre, lorsqu'on augmente le nombre de lectures ou d'écritures dans la boucle, la localité des références réduit le temps des communications et converge pour un nombre *n* d'opérations. Sur le graphique, la convergence survient après environ 300 opérations bouclées. Le tableau A.1 montre les temps des lectures et écritures, avec ou sans FIFO, pour 1 puis pour *n* opérations. Les résultats de lectures de Syslib sont présentés à la figure A.1a puis les écritures à la figure A.1b. Nous observons des résultats similaires pour SystemC. Nous présentons les résultats des lectures à la figure A.2 et des écritures à la figure A.3. Pour chaque cas, nous présentons le temps des opérations obtenus en utilisant des *sc_fifos* et des *sc_buffers* (sans FIFO). Les mesures d'écriture avec SystemC ne tiennent pas compte de l'appel de la méthode *request_update()* nécessaire pour compléter l'opération d'écriture [RHGK01]. De plus, nous remarquons sur toutes ces figures des erreurs de mesures (les pics) causées par des changements de contexte liés au système d'exploitation.



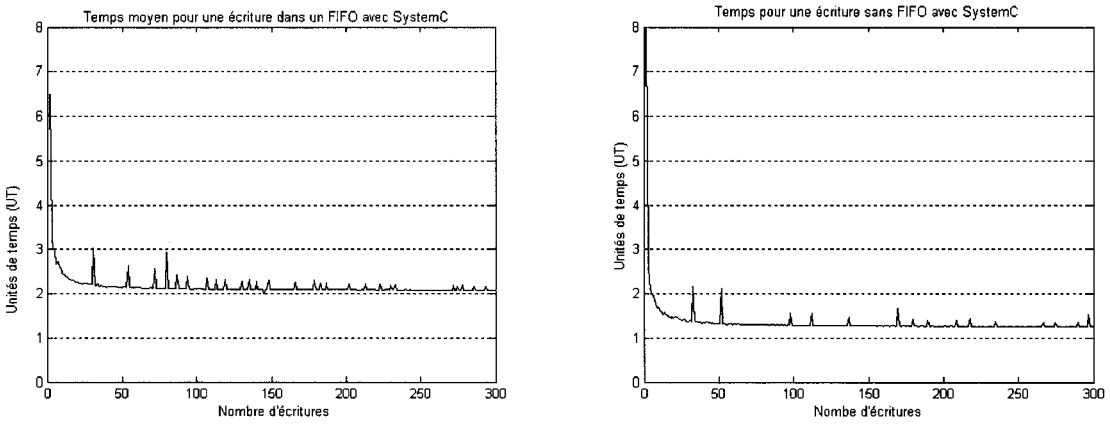
(a) (b)

Figure A.1 : Temps moyens pour des lectures et écritures avec Syslib



(a) (b)

Figure A.2 : Temps moyens pour des lectures avec SystemC



(a) (b)

Figure A.3 : Temps moyens pour des écritures avec SystemC

Tableau A.1: Performance des lectures/écritures pour Syslib et SystemC

Opération	Temps moyen pour 1 opération (*UT*)	Temps moyen pour *n* opérations (*UT*)
Syslib lecture	6,0	0,71
Syslib écriture	5,0	0,67
SystemC sans FIFO lecture	8,0	0,93
SystemC avec FIFO lecture	7,0	2,07
SystemC sans FIFO écriture	8,0	1,25
SystemC avec FIFO écriture	6,5	2,07

On peut prendre note que ces tests auraient pu être effectués avec un outil existant au laboratoire de recherche du nom de « rdtsc ».

ANNEXE B

Résultats du décodeur JPEG

Plusieurs tests ont été effectués avec le décodeur JPEG pour vérifier divers éléments. Le tableau B.1 affiche les résultats complets de ces tests.

Tableau B.1 : Résultats complets pour le décodeur JPEG

	Temps d'exécution (*UT*)			Écart de performance		Fichiers (ko)	
Fichier	SystemC	Syslib avec hiérarchie	Syslib sans hiérarchie	avec hiérarchie	sans hiérarchie	JPEG	Bitmap
128x128.jpg	268627	235927	170836	13.86%	57.24%	6	49
320x240.jpg	1240823	1096788	795922	13.13%	55.90%	41	226
384x480.jpg	2891514	2574706	1871207	12.30%	54.53%	38	541
400x592.jpg	3758112	3354695	2422679	12.03%	55.12%	86	694
416x608.jpg	4041546	3575819	2614552	13.02%	54.58%	111	742
1024x640.jpg	10403852	9254340	6748732	12.42%	54.16%	185	1921
2048x1536.jpg	49474461	44060864	31631216	12.29%	56.41%	286	9217
4096x3072.jpg	207860712	175690312	127480720	18.31%	63.05%	895	36865
tiled256x256.jpg	1062557	930159	672877	14.23%	57.91%	25	193
nottiled256x256.jpg	1058850	936894	683404	13.02%	54.94%	36	193
tiled1024x1024.jpg	16593439	14910269	10833364	11.29%	53.17%	377	3073
nottiled1024x1024.jpg	16544476	14882800	11012196	11.17%	50.24%	352	3073

Les tests ont été menés en utilisant différents fichiers de tailles croissantes. Pour chaque fichier, nous mesurons le temps d'exécution en utilisant le compteur de performance de Windows. Le temps est mesuré pour des simulations avec SystemC, puis avec la version hiérarchique et non hiérarchique de Syslib. Nous mesurons, pour les deux versions de la bibliothèque Syslib, l'écart de performance par rapport à SystemC. Enfin, nous présentons à titre indicatif les tailles des fichiers d'entrée (JPEG) et de sortie (Bitmap).

Le plus petit fichier est de résolution 128 pixels par 128 pixels et nous augmentons progressivement la résolution à 4096 pixels par 3072 pixels. Pour cette série de fichiers, nous remarquons que Syslib est plus performant (que SystemC) de 11% à 18% si on utilise la version hiérarchique. La version non hiérarchique est plus performante (que

SystemC) de 54% à 63%. Les versions Syslib sont significativement plus performantes pour de gros fichiers.

Les quatre derniers fichiers présentent un test particulier dans lequel nous voulions vérifier l'effet de la localité des références du décodeur. Nous utilisons deux fichiers de même taille, mais un de ces fichiers contient un patron de 64 pixels par 64 pixels répété en tuile (*tiled*) comme le montre la figure B.1.

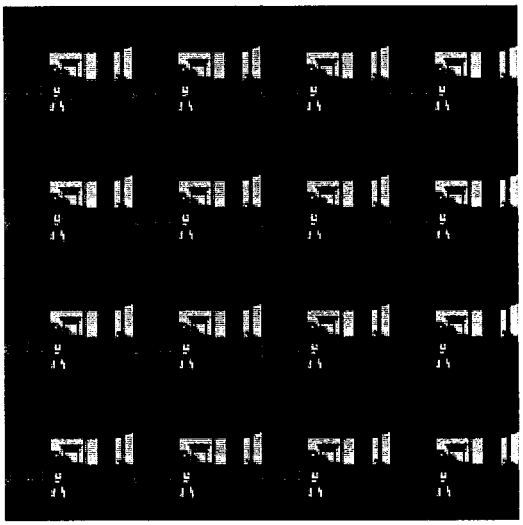


Figure B.1 : Image avec un patron tuilé

En observant les résultats, nous remarquons que la localité n'influence pas le temps d'exécution. L'écart de performance est légèrement plus faible, mais d'autres tests devraient être effectués pour porter une conclusion sur ce phénomène.

ANNEXE C

Exécution pipelinée du décodeur JPEG

Cette annexe apporte une analyse du pipeline d'exécution du décodeur JPEG version Syslib. Le traitement du décodeur peut être pipeliné puisque tous les étages d'exécution sont balancés. Un étage est une unité d'exécution pouvant être parallélisée aux autres étages. Dans le cas du décodeur JPEG, nous identifions 5 étages d'exécution : l'envoi des données de l'image vers le décodeur (module *M_PROC*), la décompression de Huffman (module *M_IHUFFMAN*), la quantification inverse (module *M_IQUANT*), la DCT inverse (module *M_IDCT*) et la collecte des données décompressées (module *M_PROC*). L'exécution pipelinée parfaite est démontrée à la figure C.1 où les blocs de données à traiter remplissent les étages du décodeur. Pour plus d'informations sur le décodeur JPEG, référez-vous au chapitre 4.

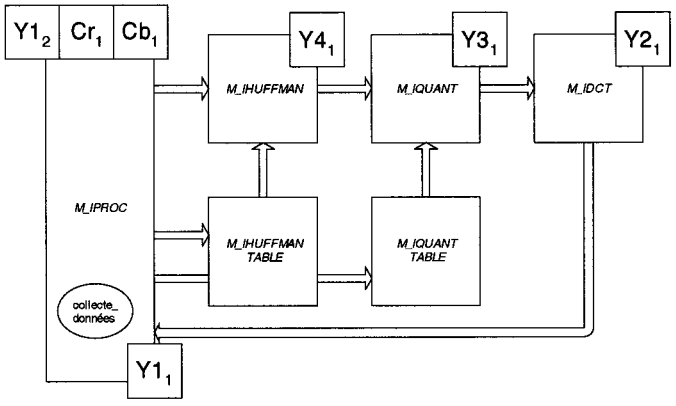


Figure C.1 : Pipelinage parfaitement balancé du décodeur JPEG

C.1. Analyse du pipeline

Voyons comment ce pipeline se comporte dans la réalité. La trace d'exécution utilise les raccourcis expliqués au tableau C.1.

Tableau C.1 : Raccourcis utilisés pour la trace du pipeline

Raccourci	Description
PROC0	Module *M_PROC* envoie les tables
PROC1 ou P1	Module *M_PROC* envoie les données de l'image
PROC2 ou P2	Module *M_PROC* collecte les données décompressées
HUFF ou H	Module *M_IHUFFMAN* en exécution
HUFFTBL	Module *M_IHUFFMANTABLE* en exécution
QTZ ou Q	Module *M_IQUANT* en exécution
QTBL	Module *M_IQUANTTABLE* en exécution
DCT ou D	Module *M_IDCT* en exécution

C.1.1. Séquence de lecture des tables

Au début de l'exécution, la lecture de l'image est effectuée. De cette image sont extraites les tables de quantification inverse et les tables de Huffman. Ces opérations ne sont que temporaires et ne peuvent donc pas faire parti du pipelinage. L'ordonnancement des modules de cette partie est présenté à la figure C.2.

```
PROC0
QTBL
PROC0
QTBL
PROC0
HUFFTBL
PROC0
HUFFTBL
PROC0
HUFFTBL
PROC0
HUFFTBL
```

Figure C.2 : Séquence de lecture des tables

C.1.2. Séquence de lecture de l'image

La figure C.3 présente la trace d'exécution du décodeur par bloc de pixels (ou MCU). L'ordre d'ordonnancement des modules est présenté sur l'axe vertical.

t	MCU1	MCU2	MCU3	MCU4
	PROC1			
	HUFF (table dc)			
	HUFFTBL			
	HUFF (table ac)			
	HUFFTBL			
	QTZ (table)			
		PROC1		
	QTBL			
		HUFF		
	QTZ (opération)			
		HUFFTBL		
	DCT			
		HUFF		
	PROC2			
		HUFFTBL		
		HUFF		
		QTZ (table)		
			PROC1	
		QTBL		
			HUFF	
		QTZ (opération)		
			HUFFTBL	
		DCT		
			HUFF	
		PROC2		
			HUFFTBL	
			HUFF	
			QTZ	
				PROC1
			QTBL	
				HUFF
			QTZ (opération)	
				HUFFTBL
			DCT	
				HUFF
			PROC2	
				...

Figure C.3 : Séquence de décompression de l'image

Cette table peut être reprise dans un diagramme d'exécution pipelinée tel que démontré à la figure C.4. Dans ce schéma, seuls les étages actifs sont conservés (i.e. on retire les accès aux tables). On constate que le pipelinage réel n'est pas si parfait, puisque les opérations ne prennent pas le même nombre d'ordonnancement. Les étages P1, P2 et D s'exécutent en un seul ordonnancement, mais à cause des accès aux tables, l'étage H demande 4 ordonnancements et Q demande 3 ordonnancements.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MCU1	P1	H	H	H	H	Q	Q	Q	D	P2						
MCU2						P1	H	H	H	H	Q	Q	Q	D	P2	
MCU3											P1	H	H	H	H	Q

Figure C.4 : Diagramme pipeliné pour la lecture de l'image

On peut démontrer après analyse qu'une optimisation de l'exécution est possible en ramenant l'opération P1 pendant que l'opération H est en cours. La figure C.5 montre que cette optimisation permet d'économiser 1 ordonnancement par bloc de base (MCU) à traiter. Cette optimisation est possible en ajoutant des signaux de contrôle adéquat.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MCU1	P1	H	H	H	H	Q	Q	Q	D	P2						
MCU2		P1				H	H	H	H	Q	Q	Q	D	P2		
MCU3						P1				H	H	H	H	Q		

Figure C.5 : Diagramme pipeliné optimisé pour la lecture de l'image

ANNEXE D
Transformation de code automatisée

La méthodologie de codesign intégrant Syslib dicte que les modules partitionnés en matériel doivent être raffinés vers la bibliothèque de modélisation matérielle Cynlib. Or, cette transformation doit être faite manuellement et ouvre la porte à l'introduction d'erreurs fortuites au cours du raffinement progressif. Cette annexe explique les grandes lignes d'un outil créé à partir d'un analyseur syntaxique pour transformer automatiquement, de Syslib à Cynlib, les parties importantes de la structure des modules. Pour plus de renseignements, il faudra consulter [Klei00], [Lefr02] et [Davi00].

Pour ce travail, nous utiliserons l'analyseur syntaxique (*parser*) reconnu nonmé JFlex [Klei00]. JFlex génère du code Java qui permettra d'intégrer le transformateur à l'outil Picasso (Annexe E). Il faut d'abord créer une grammaire syntaxique qui représente la syntaxe, les éléments Syslib à transformer vers Cynlib. Tous les cas possibles de transformation doivent être préalablement identifiés afin de créer la grammaire appropriée. Il faut ensuite les programmer à l'aide d'expressions régulières qui régissent les règles d'analyse. La grammaire et le code ont été complétés par Hugo LeFrançois dans le cadre d'un projet de fin d'études (PFE) à l'École Polytechnique de Montréal [Lefr02]. Il faut prendre note que cet outil tel quel ne couvre pas tous les cas possibles, mais sert plutôt à prouver qu'on peut aisément réaliser un outil de la sorte.

D.1. Implantation

La majorité des transformations possibles auront lieu dans la structure des modules, comme il sera démontré.

D.1.1. Transformation des types de données et des ports

La première modification apportée a été la transformation des types de données. Pour les deux bibliothèques, on utilise des ports d'entrée et de sortie pour communiquer entre les modules. En Syslib, on retrouve plusieurs types de données pour créer ces ports (voir le tableau D.1 pour la liste des types). Syslib contient également le *SysEvent* qui utilise les ports pour communiquer des évènements d'un module à un autre. Du côté de Cynlib, le concept d'événement n'existe pas : Cynlib traite les évènements comme une donnée de 1 bit. Il n'existe pas non plus de types spécifiques, mais plutôt un nombre qui représente la largeur du port en bits. Les ports à transformer sont présentés au tableau D.2.

Tableau D.8 : Types de données à transformer

Type Syslib	Description	Type Cynlib correspondant
SysLong	Donnée de 32 bits	<32>
SysULong	Donnée non signée de 32 bits	<32>
SysInt	Donnée de 32 bits	<32>
SysUInt	Donnée non signée de 32 bits	<32>
SysShort	Donnée de 16 bits	<16>
SysUShort	Donnée non signée de 16 bits	<16>
SysChar	Donnée de 8 bits	<8>
SysUChar	Donnée non signée de 8 bits	<8>
SysBool	Donnée booléenne de 1 bit	<1>

Tableau D.9 : Ports à transformer

Type de port	Syslib	Cynlib
Port d'entrée	SysInPort<SysInt> name1 ;	In<32> name1 ;
Port de Sortie	SysOutPort<SysChar> name2 ;	Out<8> name2 ;

Voici maintenant les règles syntaxiques définies dans JFlex pour réaliser ces opérations :

```
<YYINITIAL>
       "SysInPort"          { out.write("In"); nbParam++;      yybegin(DATAIN); }
       "SysOutPort"         { out.write("Out"); nbParam++;   yybegin(DATAOUT); }
```

```
<DATAIN>
{
        "<SysLong>"  | "<SysULong>"  | "<SysInt>" | "<SysUInt>"
        {
              out.write("<32>");
              if(nbParam != 0)
                   parametre.append(", ");
              parametre.append("In<32> p").append(nbParam);
             yybegin(GETATTRIBUT);
        }

        "<SysShort>" | "<SysUShort>"
        {
             out.write("<16>");
             if(nbParam != 0)
                  parametre.append(", ");
             parametre.append("In<16> p").append(nbParam);
             yybegin(GETATTRIBUT);
        }

        "<SysChar>"  | "<SysUChar>"
        {
             out.write("<8>");
             if(nbParam != 0)
                  parametre.append(", ");
             parametre.append("In<8> p").append(nbParam);
             yybegin(GETATTRIBUT);
        }

        "<SysBool>"  | "<SysEvent>"
        {
             out.write("<1>");
             if(nbParam != 0)
                  parametre.append(", ");
             parametre.append("In<1> p").append(nbParam);
             yybegin(GETATTRIBUT);
        }
}
```

La première partie qui contient *SysInPort* et *SysOutPort* est dans la section *YYINITIAL*, section dans laquelle JFlex commence son traitement. Les opérations Java entre accolades seront effectuées, par exemple pour *SysInPort : out.write ("In")* et *yybegin (DATAIN)* seront exécutées. L'objet *out* est un objet *java.io.FileWriter* qui représente le fichier de sortie : le module transformé en Cynlib. La fonction *yybegin* amène l'analyseur syntaxique dans une autre section : *DATAIN* où on transforme le type de données. Par exemple, si *<SysUChar>* est détecté, la transformation dans le fichier de sortie sera *<8>*.

D.1.2. Ajout d'une horloge

Pour le bon fonctionnement des modules Cynlib, un port d'horloge sera ajouté au module en transformation. Pour ce faire, on utilise la règle qui détecte le mot clef *public*. Immédiatement après, on ajoute au fichier de sortie *In<1> clk*. Ceci assume qu'il ne doit y avoir un seul mot clef *public* lors de la déclaration des modules.

```
"public:"      { out.write(yytext());
                 out.write("\n\t In<1> clk;");
                  parametre.append("In<1> pclk"); }
```

D.1.3. Création d'un constructeur

Pour la cération de constructeurs, le nom de la classe en traitement est récupéré. Pour ce faire, nous avons besoin d'une commande qui recherche une chaîne de caractères débutant par « class » suivie de n'importe quel caractère de l'alphabet ou le caractère *souligné*. La fonction *yytext()* retourne toujours la chaîne de caractère en court de traitement.

```
"class "[a-zA-Z_0-9]+ {  className = yytext().substring(6);
                          out.write("class ");
                                        out.write(className); }
```

Le problème principal de la transformation de Syslib vers Cynlib est sans doute le constructeur. La version du constructeur de Cynlib inclut des *sockets* en paramètres pour chacun des ports du module, ce qui n'existe pas dans Syslib. Il faudra respecter un certain ordre pour les ports du constructeur, cet ordre sera le même que la déclaration des ports dans le fichier. Les ports sont conservés dans une liste nommée *parametre* puis, le constructeur ne sera créé qu'une fois le module parcouru en entier. Cette liste servira également à la création du constructeur dans le fichier d'implantation (.CPP).

```
"SysAddBehaviour"     { yypushback(15);
                                out.write("}\n\n");
                                out.write(className);
                                out.write("::");
                        out.write(className);
                                out.write("(");
                                out.write(parametre.toString());
                        out.write(")\n{\n");
                                out.write("\tclk.bind(pclk);\n");
                        for( i = 0; i < nbParam; i++)
                        {
                               out.write(attribut[i].toString());
                               out.write(".bind(");
                               out.write("p");
                               out.write(Integer.toString(i+1));
                               out.write(");\n");
                        };
                                yybegin(CONSTRUCTEUR); }
```

D.1.4. Transformation des fonctions de rappel

Il faut maintenant transformer les fonctions de rappel en Cynlib. Les 3 états suivants effectuent ce travail.

```
<CONSTRUCTEUR>
{
        "SysAddBehaviour("      {  out.write("\n\texecute_on(");
                                                    nbArgu = 0;
                                                    yybegin(EXECUTEON); }
        "}"                     { out.write(yytext());
                                                    yybegin(YYINITIAL);}
        [^]                     { out.write(yytext()); }
}
```

```
<EXECUTEON>
{
        ","             { out.write(yytext());
                                    nbArgu++;
                                    yybegin(ARGUMENT); }
        ");"            { out.write(yytext());
                                    yybegin(CONSTRUCTEUR); }
        [^]             { out.write(yytext()); }
}
```

```
<ARGUMENT>
{
        ","             { yypushback(1);
                                yybegin(EXECUTEON); }
        "0"             { if(nbArgu == 1)
                                    {
                                            out.write("\"+\", &clk, 0);");
                                            yybegin(CONSTRUCTEUR);
                                     }
                                    else
                                            out.write(yytext()); }
        ");"            { out.write(yytext());
                                    yybegin(CONSTRUCTEUR); }
        "&"             { if(nbArgu == 1)
                                    {
                                           yypushback(1);
                                           out.write("\"=\",");
                                           nbArgu++;
                                     }
                                     else
                                           out.write(yytext()); }
        [^]             { out.write(yytext()); }
}
```

En premier lieu, l'état *CONSTRUCTEUR* cherche une fonction *SysAddBehaviour*, qui est remplacée par son homologue Cynlib *execute_on*. Ensuite, on initialise le nombre d'arguments de *execute_on* à 0 et on passe à l'état *EXECUTEON*. Dans cet état, on recherche les virgules qui signifient qu'on arrive à l'argument suivant qu'on règle à l'état *ARGUMENT*. Pour mieux comprendre le principe de tri des arguments voici un exemple des 2 types de *SysBehaviour* que l'on retrouve en Syslib (1 et 2), suivis de leur transformation en Cynlib (3 et 4). Toutes les transformations des *SysBehaviour* perpétuels

en Cynlib utiliseront une horloge et seront, par conséquent, synchrones, d'où l'ajout des *"+", &clk*. Pour les autres fonctions, il faut rajouter *"="* en Cynlib comme premier paramètre.

```
1.      SysAddBehaviour(&NetworkCard::pollNetwork, 0);
2.      SysAddBehaviour(&NetworkCard::resend, &pResend, 0);

3.      execute_on(&NetworkCard::pollNetwork, "+", &clk, 0););
4.      execute_on(&NetworkCard::resend, "=",&pResend, 0);
```

D.1.5. Transformation des communications

En Syslib, les ports utilisent des méthodes d'accès, ce qui n'existe pas en Cynlib. Par exemple en Syslib, il faut affecter une valeur au port et ensuite appeler la fonction *write* pour transmettre la valeur au canal. La solution retenue pour la transformation Cynlib a été de créer, pour chaque port de sortie de données, un deuxième port d'un seul bit qu'on utilisera comme un booléen pour confirmer une nouvelle valeur dans le port contenant la donnée à transférer. Pour ce qui est des fonctions sur les ports d'entrées, en Syslib il en existe trois : *peek, consume* et *read*. Les deux premières sont des fonctions réservées aux événements. En Cynlib, la fonction *consume* perd totalement son sens, tandis que la fonction *peek* doit être remplacée par *== 1*, parce qu'elle est toujours utilisée dans un *if* pour vérifier si un événement est actif. La fonction *read* perd également son utilité en Cynlib puisque la valeur reçue n'a pas besoin d'être extraite d'un canal.

La partie la plus problématique de la conversion entre Syslib et Cynlib est certainement la conversion des *SysChannel*. Un canal possède une profondeur qui est ajustable par une méthode externe. En Cynlib, on connecte 2 ports ensembles sans utiliser de canaux (par les *sockets*). Il n'y a pas de possibilités à même la bibliothèque pour créer un FIFO entre les 2 ports. Il faut programmer un FIFO de toute pièce. Dans le cas de cette transformation, nous retirons simplement les canaux.

D.2. Compilation et exécution

Les spécifications de Syslib sont transformables en Cynlib sous les conditions suivantes. Premièrement, il faut installer la version 1.1.8 (ou mieux) de *Java Development Kit* (*JDK*). Il faut également compiler la grammaire complète (*SysToCyn.flex*) à l'aide de JFlex en fichier Java (*SysToCyn.java*). Il s'agit ensuite d'inclure et d'appeler le résultat (qui constitue une classe Java) de la façon suivante.

```
public static void main(String argv[] ) throws java.io.IOException
{
  if (argv.length == 0) {
    System.out.println("Usage : java SysToCyn <inputfile>");
  }
  else {
    for (int i = 0; i < argv.length; i++) {
      SysToCyn scanner = null;
      try {
            out = new java.io.FileWriter(nomFichier.append(argv[i]).toString());
        scanner = new SysToCyn( new java.io.FileReader(argv[i]) );
        while ( !scanner.yy_atEOF ) scanner.yylex();
      }
      catch (java.io.FileNotFoundException e) {
        System.out.println("File not found : \""+argv[i]+"\"");
      }
      catch (java.io.IOException e) {
        System.out.println("IO error scanning file \""+argv[i]+"\"");
        System.out.println(e);
      }
      catch (Exception e) {
        System.out.println("Unexpected exception:");
        e.printStackTrace();
      }
    }
    out.close();
  }
}
```

Pour l'exécution, il suffit d'entrer dans la ligne de commande : *java SysToCyn module.cpp*, où *module.cpp* est le module Syslib à transformer. Le fichier d'en-tête doit être dans le même répertoire que son implantation. Le fichier de sortie fusionnera les deux fichiers Syslib (en-tête en implantation) et portera le même nom que ces derniers, précédé du préfixe *Cyn* (*Cynmodule.cpp* dans notre cas).

D.3. Exemples d'exécution

Voici l'exemple d'un module Syslib (en-tête et implantation) en Syslib puis le résultat transformé en Cynlib.

```
#ifndef NetworkCard_H
#define NetworkCard_H

#include "Syslib.h"

class NetworkCard : public SysModule {
public:

        NetworkCard();
        ~NetworkCard();
        void pollNetwork();
        void resend();

        SysInPort<SysInt>       pFromNetwork;
        SysOutPort<SysInt>      pToNetwork;

        SysOutPort<SysInt>      pFifo;
        SysInPort<SysInt>       pPacket;

        SysInPort<SysEvent>     pResend;
        SysOutPort<SysEvent> pFull;
        SysOutPort<SysEvent> pHalf;

};
#endif //NetworkCard_H
```

Fichier d'en-tête Syslib *module.h*

```
NetworkCard::NetworkCard()
{
        SysAddBehaviour(&NetworkCard::pollNetwork, 0);
        SysAddBehaviour(&NetworkCard::resend, &pResend, 0);
}
void NetworkCard::resend()
{
        if (pResend.peek())
        {
                printf("NetworkCard::resend will redirect to network value: ");

                // consume event
                pResend.consume();

                // retreive last packet from channel
                pPacket.read();
                pToNetwork = pPacket;
                printf("%d\n", pToNetwork.value());
                pToNetwork.write();
        }
        getchar();
}
```

Fichier d'implantation Syslib *module.cpp*

```
#ifndef NetworkCard_H
#define NetworkCard_H

#include "Cynlib.h"

class NetworkCard : public CynModule {

public:
         In<1> clk;

        NetworkCard();
        ~NetworkCard();
        void pollNetwork();
        void resend();

        In<32>          pFromNetwork;
        Out<32> pToNetwork;
        Out<1>   putpToNetwork;

        Out<32> pFifo;;
        Out<1>  putpFifo;
        In<32>          pPacket;

        In<1>           pResend;
        Out<1>  pFull;
        Out<1>  pHalf;

        NetworkCard(In<1> pclk, In<32> p1, Out<32> p2, Out<1> p3, Out<32> p4, Out<1> p5,
In<32> p6, In<1> p7, Out<1> p8, Out<1> p9);
};
#endif //NetworkCard_H

NetworkCard::NetworkCard(In<1> pclk, In<32> p1, Out<32> p2, Out<1> p3, Out<32> p4, Out<1>
p5, In<32> p6, In<1> p7, Out<1> p8, Out<1> p9)
{
        clk.bind(pclk);
        pFromNetwork.bind(p1);
        pToNetwork.bind(p2);
        putpToNetwork.bind(p3);
        pFifo.bind(p4);
        putpFifo.bind(p5);
        pPacket.bind(p6);
        pResend.bind(p7);
        pFull.bind(p8);
        pHalf.bind(p9);
        execute_on(&NetworkCard::pollNetwork, "+", &clk, 0););
        execute_on(&NetworkCard::resend, "=",&pResend, 0);
}
void NetworkCard::resend()
{
        if (pResend == 1)
        {
                //printf("NetworkCard::resend will redirect to network value: ");

                // retreive last packet from channel
                pToNetwork = pPacket;
                //printf("%d\n", pToNetwork.value());
                putpToNetwork <<= 1;
        }

}
```

Fichier résultant Cynlib *Cynmodule.cpp*

ANNEXE E

Picasso

Pour spécifier tout système, l'entrée de données est beaucoup plus efficace et intéressante si on propose une interface graphique qui allège le travail des concepteurs. Pour les besoins du groupe de recherche, nous avons créé l'outil Picasso. Picasso sert à l'entrée de spécifications des systèmes embarqués et à la génération de la structure du système résultant. Techniquement, Picasso est programmé en Java et utilise la bibliothèque SWING pour son interface graphique. Java est avantageux sous plusieurs points. D'abord, il possède des bibliothèques complètes de structures de données et de services efficaces et simples à utiliser. Par ailleurs, Java est indépendant de la machine utilisée ce qui accroît sa portabilité sur différentes plate-forme (par exemple Windows, Unix, Linux).

Cette annexe est un résumé des possibilités de Picasso et des travaux réalisés. Pour des informations complémentaires sur Picasso, consultez les documents [HBAB99a], [HBAB99b], [HFBA01] et [FiHe01].

E.1. Objectif et historique

L'objectif global de l'outil Picasso consiste au support de la méthodologie de codesign du CIRCUS dans son ensemble. Mis à part le support pour l'entrée des spécifications, l'outil Picasso doit supporter le partitionnement, la création d'une architecture configurable, l'association des spécifications aux composants de l'architecture puis la génération de la structure à simuler. Une distinction claire entre des spécifications de logiciel, de matériel ou mixtes doit être tangible. La création de *testbenchs* doit aussi être rendue possible par l'outil.

La première version de Picasso a été développée en C++ sous Microsoft Windows. À l'origine, Picasso intégrait des outils d'entrée de spécifications de matériel en VHDL combinées à des spécifications de logiciel en C. Une génération de code vers une plate-forme prédéfinie avait lieu. Picasso a par la suite migré vers Java. Les concepts offerts pour la première version Java permettaient de modéliser une spécification de matériel avec VHDL, Cynlib ou un mélange des deux. Aucun support pour le logiciel n'était fourni. Ces versions de Picasso ont été développées par Yannick Héneault dans le cadre de travaux de recherche à l'École Polytechnique [HBAB99a], [HBAB99b].

E.2. Présentation de l'outil

Les fonctionnalités de l'outil Picasso ont maintenant été étendues pour supporter des éléments supplémentaires. Comme les spécifications définies par Picasso doivent être partitionnées en fonction de l'architecture BasicARM [Hene01], un mode de création d'architecture a été implanté. Puisque l'architecture cible de la méthodologie a changé en cours de route (elle est passé de BasicARM à TarP), tout l'aspect de génération du code de l'architecture n'est pas terminé. Il aurait fallu attendre que les composants de la nouvelle architecture soient décidés et fixés, ce qui a pris plus de temps que prévu. Également, la partie de l'entrée des spécifications a été modifiée pour supporter des systèmes programmés avec Syslib. Ainsi, on peut maintenant entrer une spécification en Syslib puis générer la structure hiérarchique de la spécification pour une simulation éventuelle.

E.2.1. Entrée des spécifications

L'outil Picasso confère à un concepteur deux modes d'opération pour l'entrée des spécifications. Il est d'abord possible de créer des systèmes orientés matériel, composés de modules programmés en VHDL et/ou Cynlib. La figure E.1 présente l'interface de base de Picasso (que nous décrirons par le fait même) sur lequel on peut apercevoir un système composé de plusieurs modules.

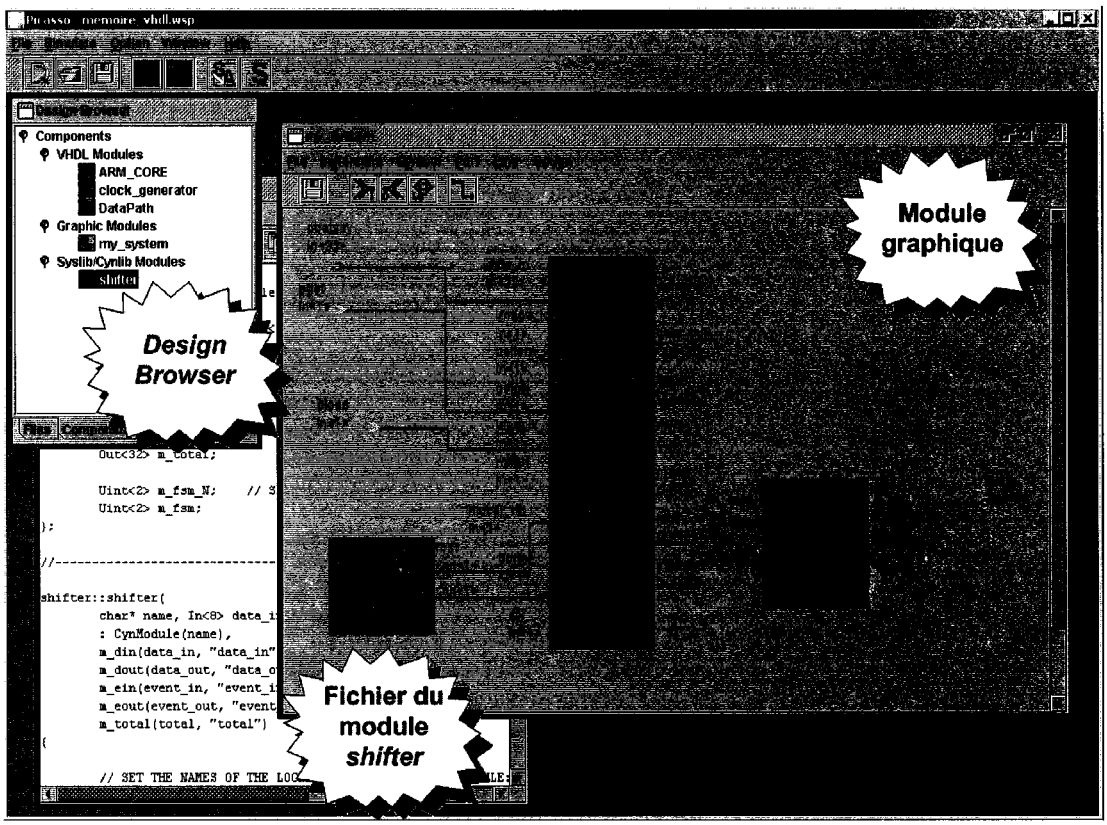


Figure E.15 : Entrée de spécifications VHDL-Cynlib avec Picasso

Grâce à l'interface, on manipule les modules du projet à l'aide de la fenêtre *Design Browser* (coin supérieur gauche). Les modules existants peuvent être listés dans le *Design Browser* par catégorie. Pour pouvoir instancier les modules, le concepteur crée un nouveau document (ou module) de type *graphique* qui permet de construire le système à partir des modules existants. La fenêtre du coin supérieur droit décrit un système composé de trois modules, dont un en Cynlib et deux en VHDL. La hiérarchie de modules est possible en créant simplement des instances de modules graphiques. En arrière plan en bas à gauche, on voit le fichier source du module *shifter* qui peut être édité et corrigé au besoin.

De la même façon, il est possible de créer une spécification en Syslib. Il est à noter qu'il n'est pas permis de mélanger des modules Syslib à des modules Cynlib ou VHDL (le système ainsi créé serait inconsistant). La figure E.2 montre un exemple de module Syslib. On y voit la création d'un nouveau port.

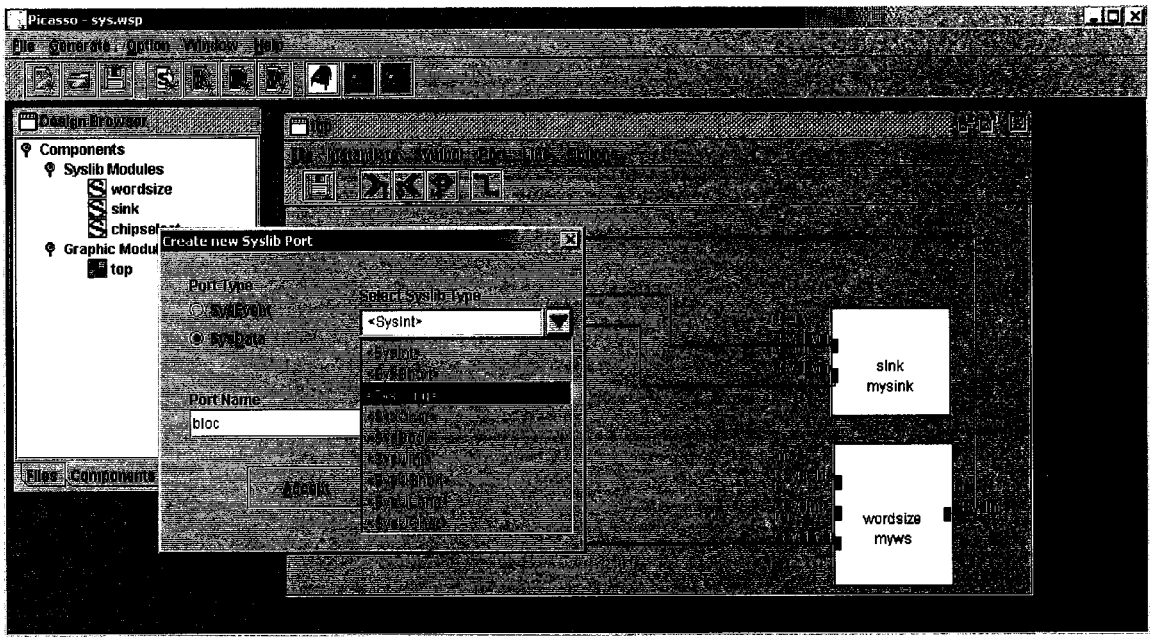


Figure E.16 : Création d'un système en Syslib

E.2.2. Mode création d'architecture

Picasso utilise une approche orientée plate-forme pour la création de SoC. À l'origine, nous avons choisi l'architecture BasicARM pour physiquement implanter les spécifications. Les composants de cette architecture sont connus et on peut par conséquent les instancier et les interconnecter pour créer de toute pièce une architecture adaptée aux besoins de l'application. Comme le démontre la figure E.3, il est possible de créer une architecture en utilisant, par exemple, un bloc processeur de base relié à une mémoire partagée et à un registre spécialisé. Chacun des composants est configurable et une boîte de dialogue simple permet d'en fixer les paramètres.

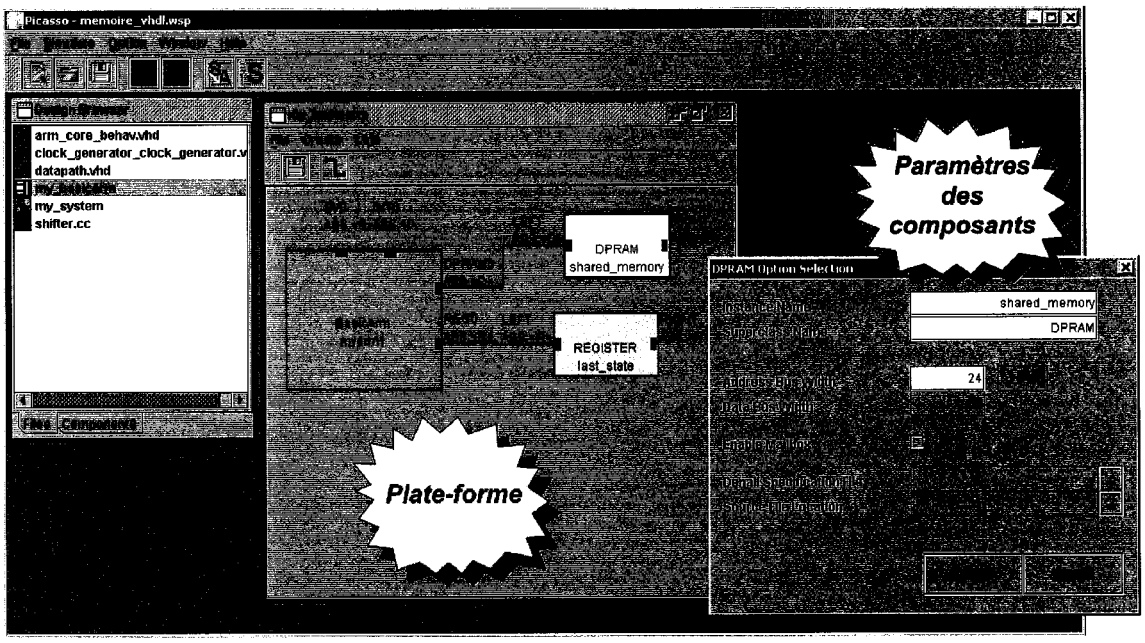


Figure E.17 : Création d'architecture avec Picasso

Le système supporte plusieurs bloc processeurs et on peut les relier entre eux à l'aide de signaux d'interruptions et d'événements.

E.2.3. Partitionnement

Une fois les paramètres de l'architecture ajustés et la spécification terminée, on peut procéder au partitionnement du système, pour la génération finale du code. Ainsi, les modules VHDL seront automatiquement implantés en matériel, alors que les modules Cynlib offrent le choix d'une implantation de matériel et de logiciel. Il faudra bien sûr programmer le module Cynlib en conséquence.

Pour chaque module Cynlib, on choisit à même l'interface le paradigme d'implantation (logiciel ou matériel) du module. On aura ainsi une kyrielle de modules matériel et logiciel qui communiquent entre eux. On devra identifier (sur les connexions) un moyen physique (d'après la plate-forme) pour que ces communications puissent se réaliser : les mémoires partagées et les registres pour les données et les interruptions ou les événements pour les signaux de contrôle. En identifiant tous les modules et toutes les

communications, on en arrive à un système complètement partitionné. Il est à noter que cette procédure supporte une ancienne vision de la méthodologie sous laquelle les modules logiciels étaient implantés avec une version modifiée de la bibliothèque Cynlib. Des travaux futurs permettront d'ajuster le partitionnement en utilisant les spécifications Syslib.

E.2.4. Génération

L'outil Picasso est utilisé pour la génération du code représentant la structure créée, ce qui réduit énormément le nombre de lignes de code qu'un programmeur aura à entrer. En ce qui a trait à la génération, quatre cas de spécifications existent :

1. Spécification uniquement composée de modules VHDL
2. Spécification uniquement composée de module Cynlib
3. Spécification composée de modules Cynlib et VHDL
4. Spécification uniquement composée de modules Syslib

Pour les cas 1 et 3, on générera un système matériel VHDL qu'on pourra simuler avec un simulateur matériel Modelsim de Mentor Graphics. Les modules Cynlib générés seront encapsulés par une interface de programmation particulière provenant de la technologie FLI [HFBA01]. Dans le cas 2 où la spécification ne possède que des modules Cynlib, on génère une spécification Cynlib exécutable. Tous les fichiers de configuration ainsi que les *makefiles* sont générés. Enfin, dans le cas 4 où une spécification Syslib est générée, on crée la structure Syslib associée au système qu'on pourra par la suite simuler.

E.3. Implantation et travaux futurs

Picasso est un outil complexe comportant plus d'une centaine de classes. Il est hors contexte de les présenter dans ce document. Le lecteur intéressé devra consulter le document [Fili01] pour des informations sur Picasso.

Picasso est un outil qui pourra toujours évoluer. Il est particulièrement intéressant dans le contexte d'un groupe de recherche d'avoir un outil global où différents projets peuvent être connectés entre eux. Le principal travail d'extension pour Picasso consistent au support de spécifications SystemC, pour mieux supporter la nouvelle méthodologie de codesign du CIRCUS. Autre point : on devra modifier les algorithmes de création d'architecture pour se lier à la nouvelle plate-forme TarP et générer l'architecture finale. Un meilleur support pour le logiciel est requis, notamment quant à l'intégration du RTOS embarqué. Enfin, on devra supporter des générations mixtes de Syslib et de Cynlib/SystemC.

ANNEXE F

Code de l'exemple du PacketRouter

F.1. Fichier PacketRouter.h

```
////////////////////////////////////////////////////////////////////////
//
//      Copyright(c) 2001 CIRCUS/GRM/Ecole Polytechnique de Montreal
//      This file is open source for personal or educational purpose only.
//      Cannot be used without quoting of the names of the authors.
//      Website: http://www.grm.polymtl.ca/circus/en
//
//      Filename:       PacketRouter.h
//      Project:        Syslib FL v0.0.0
//
//      Author:         Luc Filion
//      Lst Modif:      Oct. 2001
//
//      Description: This file is part of the Syslib Project
//
////////////////////////////////////////////////////////////////////////
#ifndef PacketRouter_H
#define PacketRouter_H

#include "Syslib.h"

class PacketRouter : public SysModule {

public:

    PacketRouter();
    void pollNetwork();
    void resend();

    SysInPort<SysInt>  pFromNetwork;
    SysOutPort<SysInt> pToNetwork;

    SysOutPort<SysInt> pFifo;
    SysInPort<SysInt>  pPacket;

    SysInPort<SysEvent>        pResend;
    SysOutPort<SysEvent> pFull;
    SysOutPort<SysEvent> pHalf;

};
#endif //PacketRouter_H
```

F.2. Fichier PacketRouter.cpp

```
////////////////////////////////////////////////////////////////////////
//
//      Copyright(c) 2001 CIRCUS/GRM/Ecole Polytechnique de Montreal
//      This file is open source for personal or educational purpose only.
//      Cannot be used without quoting of the names of the authors.
//      Website: http://www.grm.polymtl.ca/circus/en
//
//      Filename:       PacketRouter.cpp
//      Project:        Syslib FL v0.0.0
//
//      Author:         Luc Filion
//      Lst Modif:      Oct. 2001
//
//      Description: This file is part of the Syslib Project
//
////////////////////////////////////////////////////////////////////////
#include "PacketRouter.h"
#include <stdio.h>

PacketRouter::PacketRouter()
{
    SysAddBehaviour(&PacketRouter::pollNetwork, 0);
    SysAddBehaviour(&PacketRouter::resend, &pResend, 0);
}


void PacketRouter::pollNetwork()
{
    printf("PacketRouter::pollNetwork will empty its network FIFO\n");
    printf("  Values are: ");

    int pushedvalues = 0;
    while (1)
    {
        // keep going
        if (pFromNetwork.read() == SYSCHANNEL_EMPTY)
        {
            printf("  no more values. Terminating.\n");
            getchar();
            return; // no more values to retreive, terminate
        }
        else
        {
            pFifo = pFromNetwork;
            printf("%d ", pFromNetwork.value());
        }

        if (pFifo.write() == SYS_ERROR)
        {
            pFull.notify();  // notice driver that pFifo is full
            printf("\n  notifying FULL to Driver and terminating\n");
            getchar();
            return; // terminate here
        }
        else
        {
            if ( ++pushedvalues == 5)
            {
                // notice driver that pFifo is half-full
                //if ( pHalf.notify() == SYS_ERROR);
                pHalf.notify();
                printf("\n  notifying HALF to Driver\n");


            }
        }
    }
    getchar();
}

void PacketRouter::resend()
{
    if (pResend.peek())
```

```
    {
        printf("PacketRouter::resend will redirect to network value: ");

        // consume event
        pResend.consume();

        // retreive last packet from channel
        pPacket.read();
        pToNetwork = pPacket;
        printf("%d\n", pToNetwork.value());
        pToNetwork.write();
    }
    getchar();
}
```

F.3. Fichier Network.h

```
////////////////////////////////////////////////////////////////////////
//
//      Copyright(c) 2001 CIRCUS/GRM/Ecole Polytechnique de Montreal
//      This file is open source for personal or educational purpose only.
//      Cannot be used without quoting of the names of the authors.
//      Website: http://www.grm.polymtl.ca/circus/en
//
//      Filename:       Network.h
//      Project:        Syslib FL v0.0.0
//
//      Author:         Luc Filion
//      Lst Modif:      Oct. 2001
//
//      Description: This file is part of the Syslib Project
//
////////////////////////////////////////////////////////////////////////
#ifndef NETWORK_H
#define NETWORK_H

#include "Syslib.h"

class Network : public SysModule {

public:

    Network();
    void receiveData();
    void sendData();

    SysOutPort<SysInt> pDataOut;
    SysInPort<SysInt>  pDataIn;


};
#endif //NETWORK_H
```

F.4. Fichier Network.cpp

```
////////////////////////////////////////////////////////////////////////
//
//      Copyright(c) 2001 CIRCUS/GRM/Ecole Polytechnique de Montreal
//      This file is open source for personal or educational purpose only.
//      Cannot be used without quoting of the names of the authors.
//      Website: http://www.grm.polymtl.ca/circus/en
//
//      Filename:       Network.h
//      Project:        Syslib FL v0.0.0
//
//      Author:         Luc Filion
//      Lst Modif:      Oct. 2001
//
//      Description: This file is part of the Syslib Project
//
////////////////////////////////////////////////////////////////////////
#include "Network.h"
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

Network::Network()
{
    SysAddBehaviour(&Network::receiveData, 0);
    SysAddBehaviour(&Network::sendData, 0);

    srand( (unsigned)time( NULL ) );
}

void Network::receiveData()
{
    printf("Network::receiveData will empty toNetwork FIFO\n  Values are: ");
    while (pDataIn.read() != SYSCHANNEL_EMPTY)
    {
        printf("%d ", pDataIn.value());
    }
    printf("\n");
    // and that's it!
    getchar();
}

void Network::sendData()
{
    int next = (int)rand()%16;
    printf("Network::sendData generates %d values \n  Values are: ", next);
    for (int i=0; i<next; i++)
    {
        pDataOut = (int)rand()%100;
        pDataOut.write();
        printf("%d ", pDataOut.value());
    }
    printf("\n");
    // and we terminate this way :-)
    getchar();
}
```

F.5. Fichier Driver.h

```
////////////////////////////////////////////////////////////////////////
//
//      Copyright(c) 2001 CIRCUS/GRM/Ecole Polytechnique de Montreal
//      This file is open source for personal or educational purpose only.
//      Cannot be used without quoting of the names of the authors.
//      Website: http://www.grm.polymtl.ca/circus/en
//
//      Filename:       Driver.h
//      Project:        Syslib FL v0.0.0
//
//      Author:         Luc Filion
//      Lst Modif:      Oct. 2001
//
//      Description: This file is part of the Syslib Project
//
////////////////////////////////////////////////////////////////////////
#ifndef Driver_H
#define Driver_H

#include "Syslib.h"

class Driver : public SysModule {

public:

    Driver();
    void clearFifo();

    SysInPort<SysInt>  pFifo;
    SysOutPort<SysInt> pPacket;

    SysOutPort<SysEvent> pResend;
    SysInPort<SysEvent>  pFull;
    SysInPort<SysEvent>  pHalf;

};
#endif //Driver_H
```

F.6. Fichier Driver.cpp

```
////////////////////////////////////////////////////////////////////////
//
//      Copyright(c) 2001 CIRCUS/GRM/Ecole Polytechnique de Montreal
//      This file is open source for personal or educational purpose only.
//      Cannot be used without quoting of the names of the authors.
//      Website: http://www.grm.polymtl.ca/circus/en
//
//      Filename:       Driver.cpp
//      Project:        Syslib FL v0.0.0
//
//      Author:         Luc Filion
//      Lst Modif:      Oct. 2001
//
//      Description: This file is part of the Syslib Project
//
////////////////////////////////////////////////////////////////////////
#include "Driver.h"
#include <stdio.h>

Driver::Driver()
{
    SysAddBehaviour(&Driver::clearFifo, &pFull, &pHalf, 0);
}

void Driver::clearFifo()
{
    if (pFull.peek())
    {
        pHalf.consume();
        pFull.consume();
        printf("Driver::clearFifo is clearing a FULL fifo\nValues are : ");
        while (pFifo.read() != SYSCHANNEL_EMPTY)
        {
            printf("%d ", pFifo.value() );
        }
        printf("\n\n");
        printf("Driver::clearFifo wuill notify the RESEND signal\n");

        pPacket = pFifo; // return last value
        pPacket.write();
        pResend.notify(); // notify
    }
    else if (pHalf.peek())
    {
        pHalf.consume();
        printf("Driver::clearFifo is clearing a HALF fifo\nValues are : ");
        while (pFifo.read() != SYSCHANNEL_EMPTY)
        {
            printf("%d ", pFifo.value() );
        }
        printf("\n\n");
    }
    getchar();
}
```

F.7. Fichier main.cpp

```
////////////////////////////////////////////////////////////////////////
//
//      Copyright(c) 2001 CIRCUS/GRM/Ecole Polytechnique de Montreal
//      This file is open source for personal or educational purpose only.
//      Cannot be used without quoting of the names of the authors.
//      Website: http://www.grm.polymtl.ca/circus/en
//
//      Filename:       main.cpp
//      Project:        Syslib FL v0.0.0
//
//      Author:         Luc Filion
//      Lst Modif:      Oct. 2001
//
//      Description: This file is part of the Syslib projet
//
////////////////////////////////////////////////////////////////////////

#include "Network.h"
#include "PacketRouter.h"
#include "Driver.h"
#include "SysOS.h"


int main(void)
{
    Network network;
    PacketRouter network_card;
    Driver driver;

    // network to network_card channels
    SysChannel<SysInt> cToNetwork;
    SysChannel<SysInt> cFromNetwork;
    cToNetwork.setFIFODepth(16);
    cFromNetwork.setFIFODepth(16);

    // network_card to driver channels
    SysChannel<SysInt> cPacket;
    SysChannel<SysInt> cFifo;
    SysChannel<SysEvent> cFull;
    SysChannel<SysEvent> cHalf;
    SysChannel<SysEvent> cResend;
    cPacket.setFIFODepth(1);
    cFifo.setFIFODepth(8);

    // binding from network to network_card (bind out to in)
    if ( cToNetwork.bind((SysModule*)&network_card, &network_card.pToNetwork,
(SysModule*)&network, &network.pDataIn) != SYS_OK )
        printf("Can't bind ports");
    if ( cFromNetwork.bind((SysModule*)&network, &network.pDataOut,
(SysModule*)&network_card, &network_card.pFromNetwork) != SYS_OK )
        printf("Can't bind ports");

    // binding from network_card to driver (bind out to in)
    if ( cFifo.bind((SysModule*)&network_card, &network_card.pFifo, (SysModule*)&driver,
&driver.pFifo) != SYS_OK )
        printf("Can't bind ports");
    if ( cFull.bind((SysModule*)&network_card, &network_card.pFull, (SysModule*)&driver,
&driver.pFull) != SYS_OK )
        printf("Can't bind ports");
    if ( cHalf.bind((SysModule*)&network_card, &network_card.pHalf, (SysModule*)&driver,
&driver.pHalf) != SYS_OK )
        printf("Can't bind ports");
    if ( cResend.bind((SysModule*)&driver, &driver.pResend, (SysModule*)&network_card,
&network_card.pResend ) != SYS_OK )
        printf("Can't bind ports");
    if ( cPacket.bind((SysModule*)&driver, &driver.pPacket, (SysModule*)&network_card,
&network_card.pPacket ) != SYS_OK )
        printf("Can't bind ports");

    SysOSStart();
    return 0;
}
```